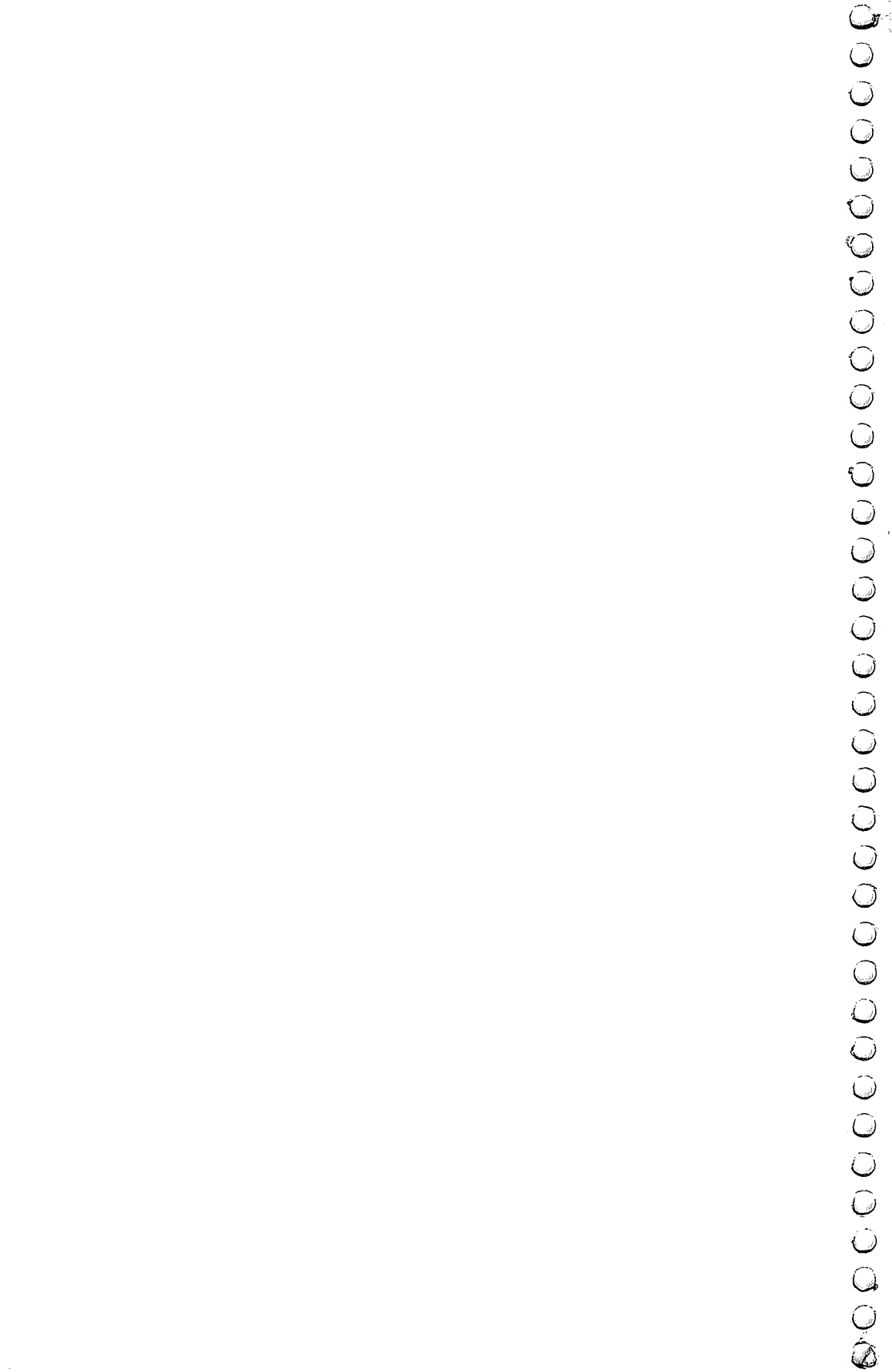


COMPUTE!'s FIRST BOOK OF

VIC

Games, Programs, And Other Helpful Information
For Owners And Users Of
The Commodore VIC-20[®] Personal Computer.





From The Editors of **COMPUTE!** Magazine

COMPUTE!'s FIRST BOOK OF VIC

Published by **COMPUTE! Books**,
A Division of Small System Services, Inc.,
Greensboro, North Carolina

VIC-20 is a trademark of Commodore Business Machines, Inc.

A
Small System
Services, Inc.
Publication

Copyright © 1982, Small System Services, Inc. All rights reserved.

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the United States Copyright Act without the permission of the copyright owner is unlawful.

"Computer Genesis: From Sticks And Stones To VIC," "Using A Joystick," "Automatic Line Numbers," and "The Window" were originally published in *Home and Educational COMPUTING!*, Fall 1981, copyright 1981, Small System Services, Inc. "Large Alphabet," *STARFIGHT3*," and "Count The Hearts" were originally published in **COMPUTE!** Magazine, March 1982, copyright 1982, Small System Services, Inc. "Extended Input Devices: Paddles And The Keyboard," "Alphabetizer," "Timekeeping," and "An Easy Way To Relocate VIC Programs On Other Commodore Computers" were originally published in **COMPUTE!** Magazine, February 1982, copyright 1982, Small System Services, Inc. "Game Paddles" and "Renumber BASIC Lines The Easy Way" were originally published in **COMPUTE!** Magazine, April 1982, copyright 1982, Small System Services, Inc. "The Joystick Connection: Meteor Maze," "Amortize," and "Putting The Squeeze On Your VIC-20: Getting The Most Out Of 5000 Bytes" were originally published in **COMPUTE!** Magazine, May 1982, copyright 1982, Small System Services, Inc. "ZAP!!," "VIC Color Tips," and "Memory Map Above Page Zero" were originally published in **COMPUTE!** Magazine, January 1982, copyright 1982, Small System Services, Inc. "Train Your PET To Run VIC Programs" was originally published in **COMPUTE!** Magazine, October 1981, copyright 1981, Small System Services, Inc. "The Confusing Quote," "Custom Characters For The VIC," and "Super Calculator" were originally published in the introductory issue of *Home and Educational COMPUTING!*, Summer 1981, copyright 1981, Small System Services, Inc. "Alternate Screens" was originally published in *Home and Educational COMPUTING!*, Fall 1981, copyright 1981, Jim Butterfield. "How To Use The 6560 Video Interface Chip" was originally published in **COMPUTE!** Magazine, July 1982, copyright 1982, Small System Services, Inc. "Browsing The VIC Chip" was originally published in **COMPUTE!** Magazine, April 1982, copyright 1982, Jim Butterfield. "TINYMON1: A Simple Monitor For The VIC" was originally published in **COMPUTE!** Magazine, January 1982, copyright 1981, Jim Butterfield.

Printed in the United States of America

ISBN 0-942386-07-8

10 9 8 7 6 5 4 3 2

v Introduction	Robert Lock
----------------------	-------------

Chapter One: Getting Started.

3 The Story Of The VIC	Michael S. Tomczyk
11 Computer Genesis:	
From Sticks And Stones To VIC	Dorothy Kunkin Heller / David Thornburg
20 Super Calculator	Jim Butterfield
24 Large Alphabet	Doug Ferguson
26 Using A Joystick	David Malmberg
39 Extended Input Devices:	
Paddles And The Keyboard	Mike Bassman / Salomon Lederman
46 Game Paddles	David Malmberg

Chapter Two: Diversions – Recreation And Education.

59 The Joystick Connection: Meteor Maze	Paul L. Bupp / Stephen P. Drop
67 ZAPI!	Dub Scroggin
72 STARFIGHT3	David R. Mizner
78 Alphabetizer	Jim Wilcox
80 Count The Hearts	Christopher J. Flynn

Chapter Three: Programming Techniques.

89 PRINTing With Style	James P. McCallister
97 Train Your PET To Run VIC Programs	Lyle Jordan
99 User Input	Wayne Kozun
103 Amortize	Amihai Glazer
106 Append	Wayne Kozun
109 Printing The Screen	C. D. Lane
113 The Confusing Quote	Charles Brannon
115 Alternate Screens	Jim Butterfield
119 Timekeeping	Keith Schleiffer
125 Renumber BASIC Lines The Easy Way	Charles H. Gould
127 Automatic Line Numbers	Jim Wilcox
129 Putting The Squeeze On Your VIC-20:	
Getting The Most Out Of 5000 Bytes	Stanley M. Berlin
141 An Easy Way To Relocate VIC Programs	
On Other Commodore Computers	Greg and Ross Sherwood

Chapter Four: Color And Graphics.

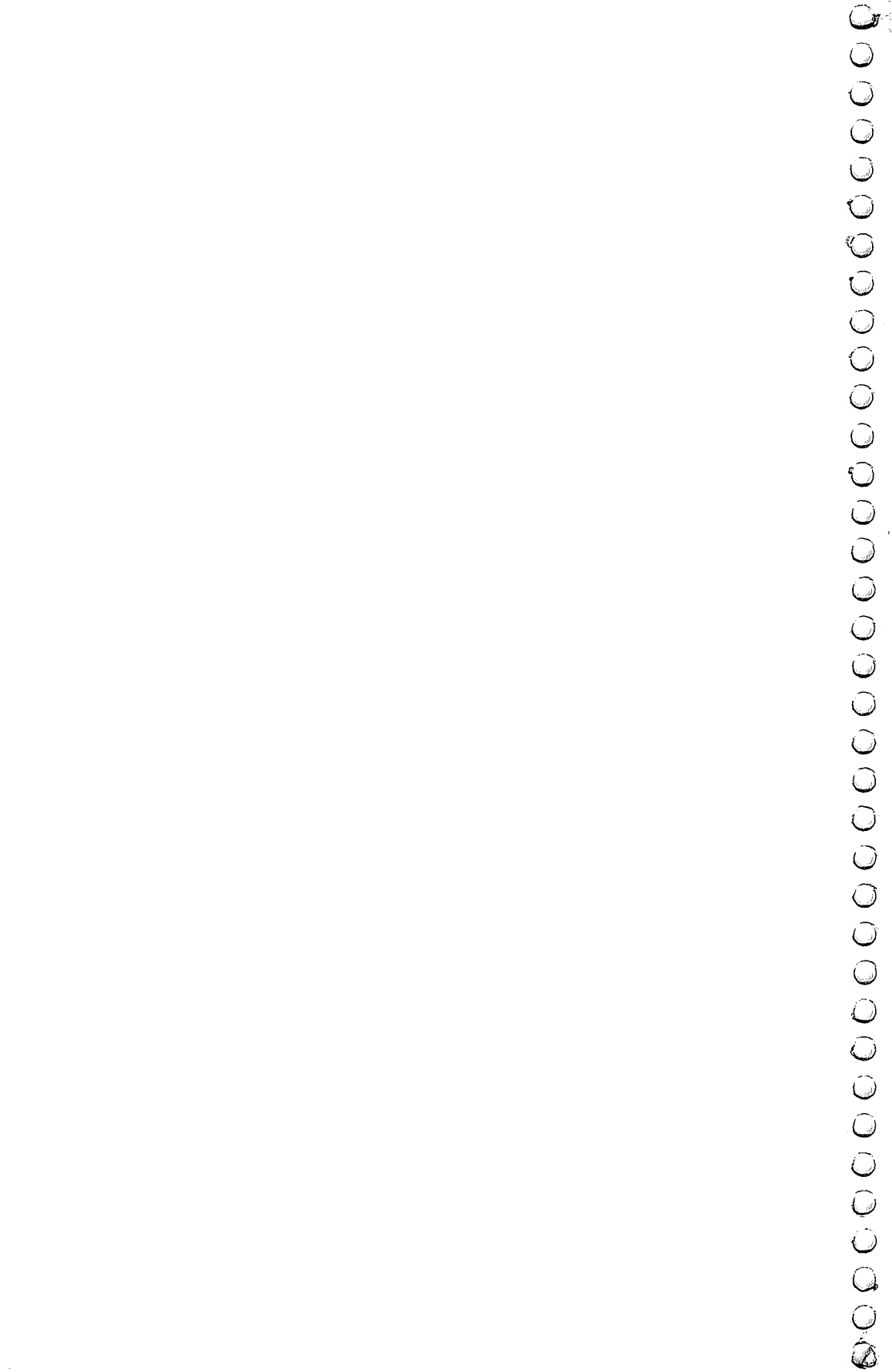
147 Kaleidoscope And Variations	Kenneth Knox
148 High Resolution Plotting	Paul F. Schatz
154 VIC Color Tips	Charles Brannon
157 The Window	Charles Brannon
160 Custom Characters For The VIC	David Malmberg

Chapter Five: Maps And Specifications.

173 How To Use The 6560 Video Interface Chip	Dale Gilbert
179 Browsing The VIC Chip	Jim Butterfield
186 VIC Memory – The Uncharted Adventure	David Barron / Michael Kleinert
189 Memory Map Above Page Zero	Jim Butterfield

Chapter Six: Machine Language.

195 TINYMON1: A Simple Monitor For The VIC	Jim Butterfield
202 Entering TINYMON1 Directly Into Your VIC-20	Russell Kavanagh
211 Index	



INTRODUCTION

Robert Lock, Editor/Publisher, **COMPUTE!** Magazine

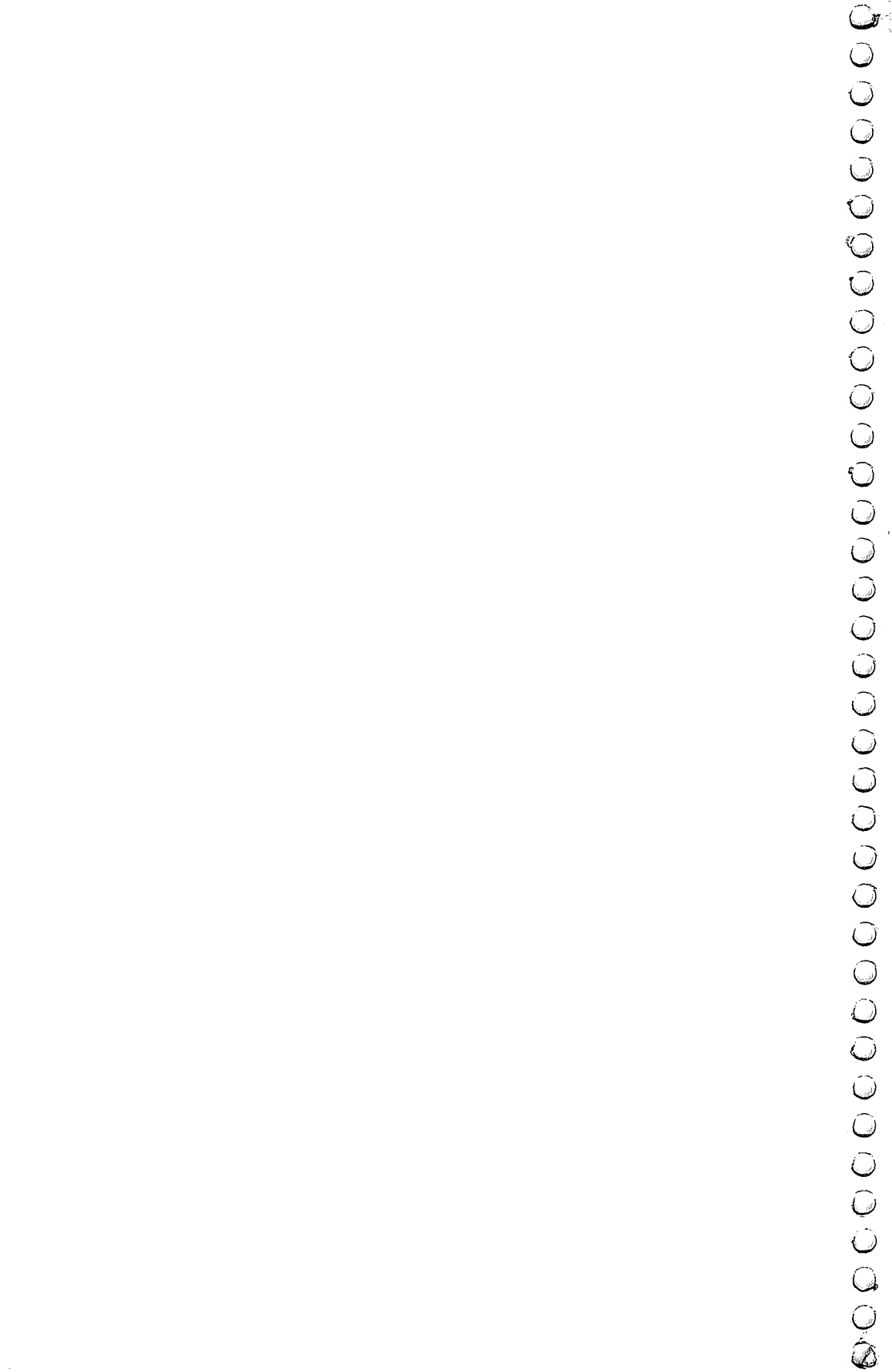
We are pleased to present this addition to our First Book Series on personal computers. In these pages you'll find many articles originally appearing in the pages of **COMPUTE!** Magazine. These articles have been carefully chosen, mixed with some previously unpublished material, and collected here for the use and enjoyment of VIC owners. Whether you're just starting with your VIC-20 personal computer, or are a far more advanced user, you'll quickly discover this book is an essential addition to your computer library.

As with our parent publication, **COMPUTE!** Magazine, you'll find a range of material, from beginner to advanced, ready to type right into your computer. Programs and helpful hints designed to teach and entice you. Applications and utilities designed to help you better utilize and explore this fascinating world of personal computing.

We've organized the material and designed the book for ease of use. We welcome your suggestions and comments on this and future titles from **COMPUTE! Books**.

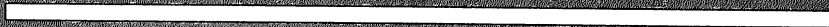
Special thanks to Charles Brannon, Richard Mansfield, Tom Halfhill, and Kathleen Martinek of our editorial staff; Kate Taylor, De Potter and Terry Cash of our typesetting and production staff; Georgia Papadopoulos, Art Director; and Harry Blair, our illustrator.

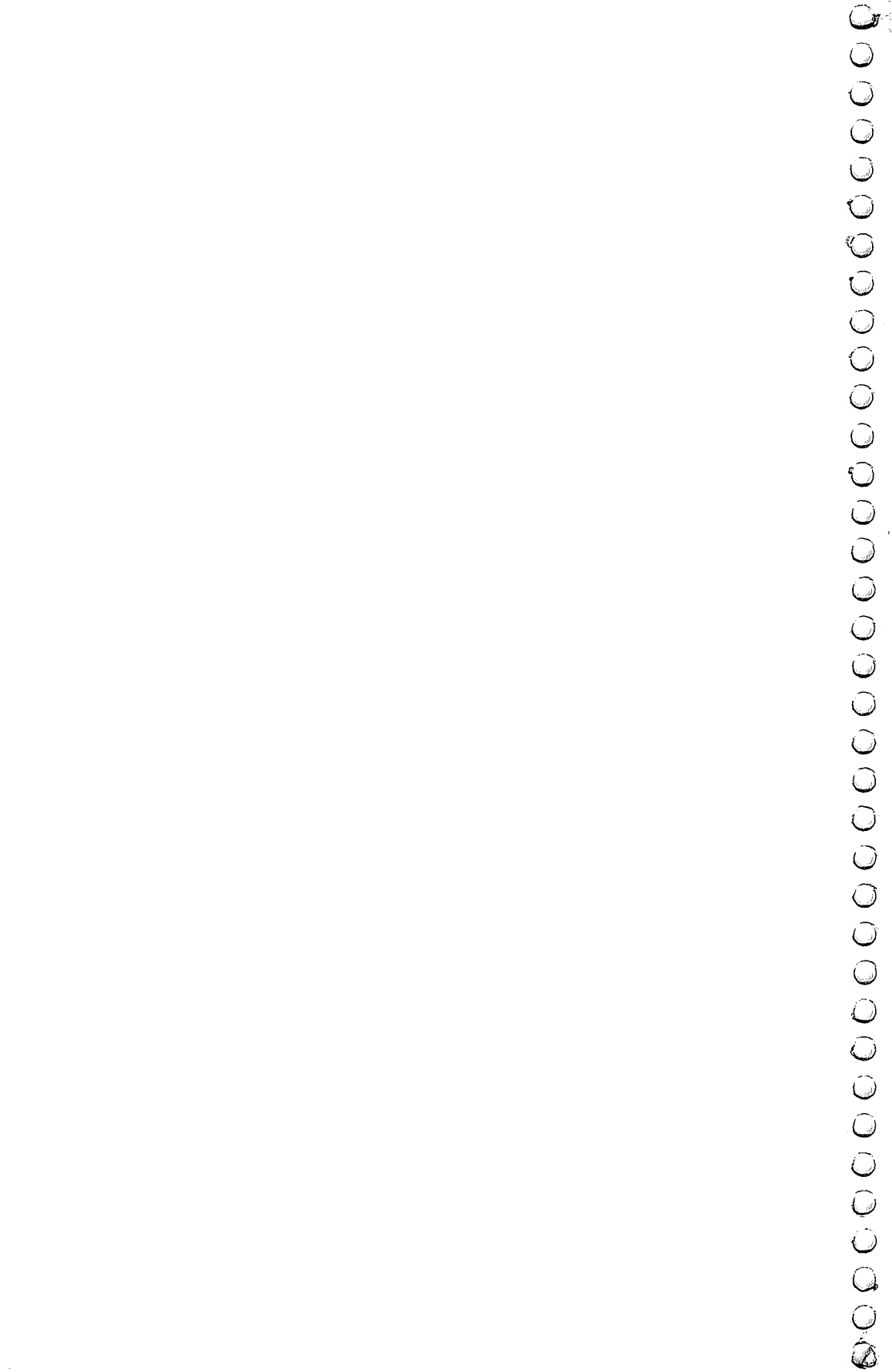
COMPUTE! Books is a division of Small System Services, Inc.,
Publishers of **COMPUTE!** Magazine.
Editorial offices are located at 625 Fulton Street, P.O. Box 5406,
Greensboro, NC 27403 USA. (919) 275-9809.



CHAPTER ONE

GETTING STARTED





The Story Of The VIC

MICHAEL S. TOMCZYK

Product Marketing Manager, Commodore International

Commodore projects 1982 sales of over 1/2 million VICs in the US alone. This compares to a total of 800,000 computers of all types sold last year in the US. Michael Tomczyk, one of the VIC's founding fathers, describes the beginnings and the history of this popular computer.

The VIC-20 was first announced by Jack Tramiel – Commodore's founder and now Vice Chairman – at an international manager's meeting in London in April 1980. There, at a quaint inn on the outskirts of the city, the representatives of half a dozen "Commodore countries" gathered to discuss problems and exchange ideas.

We Will Become The Japanese

On the second day of the meeting, Tramiel surprised everyone by announcing his intention to develop and market a "\$300 personal computer." He reminded the group that Commodore was a pioneer in low-priced pocket calculators and had introduced the first self-contained personal computer – the PET – in 1976. Now it was time to introduce a low-priced color computer.

A debate ensued, with several groups talking simultaneously. Some felt it wasn't time for a computer priced that low. Others felt the new computer might undercut sales of the PET, and still others questioned whether it was economically and technologically feasible.

Finally, about twenty minutes later, Tramiel stood up, pounded his fist once on the table, and said in his deep booming voice, "The Japanese are coming, so we will become the Japanese!" The room fell silent as he explained that several Japanese computer companies (known collectively as "Japan, Inc.") were already poised to enter the U.S. market. Japanese companies had already captured the television, radio, and small car markets, and personal computers were next on their list.

He said we have to compete with ourselves by making computers that do more and cost less, and that meant breaking the \$300 price barrier.

The First \$300 Color Computer

We knew there was price resistance at the \$300 price point: but how could Commodore make a \$300 color computer profitably? Commodore has one terrific advantage – vertical integration, which means we design and manufacture our entire product. Most important, it means we design and make our own computer chips, and computer chips are the heart of any computer.

Many people still don't know that MOS Technology (a Commodore subsidiary) developed the 6502 Microprocessor, the key computer chip used in Apple, Atari, and several other popular computers, in addition to Commodore. MOS also designed the Video Interface Chip, one of the key chips used in the VIC-20 computer.

When it was suggested that we create a computer based on the VIC chip, some of the engineers resisted the idea, claiming the chip was too "limited." They scoffed at the idea of a VIC chip computer and complained that the VIC chip allowed only 22 columns on the screen compared to PET's 40 columns.

Finally, two groups of engineers, on the East and West coasts, wound up in a race to build a prototype computer using the VIC chip. At MOS in Pennsylvania, the designer of the VIC chip spent several sleepless nights building his prototype. He put the computer in an old Commodore desktop calculator housing and used a keyboard from one of the original PET computers. The original keyboard was a calculator-style keyboard with red metallic keys. In a few days, the prototype was done.

Introducing The User-Friendly VIC

The first prototypes were taken to the National Computer Convention in Chicago in June 1980, but the new computers weren't put on display. They were set up at our booth in a room enclosed by tinted Plexiglas walls. Only a few people were allowed inside to see the new computer, but lots of noses pressed against the windows as passers-by peered in to get a glimpse of our new "secret" computer.

The next job – my job – was to put together the marketing program to launch the product in the United States. We began with a very simple premise: computers are not perceived as being "friendly," so we have to make the VIC-20 as "user friendly" as possible. A lot of people chuckled because I waved the "user friendly" banner so forcefully. Some people even resisted the idea when I dubbed the VIC "The Friendly Computer" and trademarked the phrase. But in the end, "user friendliness" turned out to be the VIC's

most important feature.

The first place I used the phrase “user friendly” was in an engineering meeting in Santa Clara, California. I began the meeting by writing, in huge block letters on a greaseboard, the words *user friendly*. I then announced quite seriously, “Anything that doesn’t meet this criterion will not be discussed in this room.”

It worked. Every time the discussion strayed, or someone suggested adding some complicated feature, I simply pointed at the board and the discussion fell right into place. It seemed that everyone knew what “user friendly” meant.

Commodore engineers all picked up on the phrase and built some very friendly computing features into the VIC-20, like two graphic symbols on each key. I insisted that we put color abbreviations on the color control keys (you can blame me for using CYAN instead of LT. BLUE), and included an L-shaped pound sign for our English friends. In Autumn 1980, I took the “user friendly” banner to Japan, where we held engineering consultations and finalized the product.

VIC...VICKIE...VIXEN...

One of the hardest challenges was giving the VIC-20 its name. In the early days, the VIC-20 really didn’t have a name.

Most of the engineers liked the name Vixen. I even doodled some sketches using a cute little fox as a logo. The name Vickie was mentioned, too, but never seriously considered. Over the next few months we considered quite a long list of names which might be acceptable internationally. We all spent long hours thumbing through our thesauri searching for an obscure but clever name like Atari or a cute name like Apple.

Finally, we decided to name the computer after the Video Interface Chip – VIC – which became Video Interface Computer. VIC sounded naked by itself, however, so we decided to add a number. But the only meaningful number was VIC-22 (based on its 22 columns). For some reason, the number 22 didn’t seem very friendly, so we settled on the name VIC-20 because the number 20 sounded “friendlier.”

Ironically, as a sidenote, we originally vetoed the name Vixen because it had undesirable connotations in German, but VIC later turned out to mean something even worse. As a result, the German model was called VC-20 and translated as “Volks Computer” (the “People’s Computer”). For awhile, the name “Volks Computer” was so well-liked that we considered using it worldwide, but the only U.S. tie-in was Volkswagen, and Volkswagens were no longer being

made. Except for Germany (VC-20) and Japan (VIC-1001), we stuck with VIC-20. In the end, the short, snappy name turned out to be easy to remember, convenient for magazine headlines, and very “user friendly.”

The Japanese Didn't Come

We first introduced the VIC not in the United States but in Japan, where the VIC's potential competition was already brewing. It was sort of like carrying coals to Newcastle, but we knew if the VIC succeeded in the Japanese market, it would succeed in the rest of the world.

Our Japanese VIC, called the VIC-1001, included uppercase English letters, PET graphics, and Japanese characters. It was introduced in September 1980 at a major computer exhibit at Seibu Department Store in downtown Tokyo. Over 100 orders were taken the first day.

When we introduced the VIC-20 in the U.S., in the spring of 1981, we still expected some low-priced Japanese computers to hit our market by Christmas. Incredibly, that didn't happen. The Japanese didn't come!

Instead, most Japanese companies ignored the low end of the market and entered the U.S. with higher priced computers in the \$2000-6000 price range. As a result, throughout 1981 and most of 1982, the Commodore VIC-20 was the first and *only* full-featured color computer priced under \$300.

A Computer Priced Like A Video Game

Personal computers had been available since the mid-1970s, but by 1980 they still hadn't become a mass market item. Three major obstacles stood in the way: 1) computers were too expensive, 2) computers weren't very “friendly”, and 3) nobody knew what to *do* with them!

The long-awaited “Home Computer Revolution” had not caught fire, and the popular use of computers was limited to hobbyists, engineers, and classrooms.

Our advertising started out comparing the VIC-20 to our closest competition, but we soon realized that there *wasn't* any competition at our price point. And those higher priced computers weren't selling very well anyway, so why compare the VIC-20 to them? Why not compare the VIC-20 to a product that *was* selling well, like video game machines?

After all, the VIC-20 was selling for the same price as a video game machine, and VIC software includes cartridge games as well

as practical programs. In other words, why buy a video game when you can buy a full-fledged computer for the same price? That message became our advertising slogan.

Actor William Shatner of "Star Trek" was chosen as our spokesperson, and in the first months of 1982 we kicked off the largest advertising campaign in Commodore's history. Shatner introduced the VIC as the "Wonder Computer of the 80s," adding, "It plays great space games, too!"

Commodore also negotiated a long-term arrangement for conversion of Bally Midway coin-operated games to cartridge – including best-sellers like *Gorf* and *Omega Race*. *Sargon II Chess* gave us one of the best chess games in personal computing, and five Scott Adams Adventure games gave us possibly the best assortment of adventure games available from a computer manufacturer. We also introduced a low-priced six-pack of games on cassette tape, with names like *Blue Meanies From Outer Space*. These games persuaded a lot of people to buy the VIC-20. The next step was to take those VIC-owners from video games to other computing. To do that, we had to cross the second obstacle to mass market computing and make computing "friendly."

Friendly Computing In Action

Owning a personal computer used to mean you had to know how to program in BASIC.

But user friendliness in marketing means *giving the customer an item that requires little or no special expertise to use, apply, or enjoy*. One way to do this is to include a really nice instruction book that lets you have fun and do interesting things without expensive peripherals or packaged software.

You don't have to be an auto mechanic to drive a car. So why should you have to be a programmer to use a computer?

The user friendly manual we wrote for the VIC-20 doesn't even *mention* the word "programming" until the last chapter. Our manual teaches you how to "compute," which we interpret as meaning "to have fun." So we talk about cartoon animation, sound and music, color graphics, and other topics. We also wrote the book so you can turn to any chapter and start computing from that point, with little or no experience. If you want to write computer music, you turn to the music chapter and start there. If you want to work with color and graphics, you start with that chapter.

What we didn't say is that if you work through the book, you'll learn how to program in BASIC, by osmosis, since most of our

examples included a very subtle introduction to programming. It was a sneaky – but helpful – way to ease new computer owners into the fundamentals of computer programming, and it meant new VIC owners had an excellent head start if they decided they wanted to learn computer programming. We also wrote a technical manual for programmers, called the *VIC-20 Programmer's Reference Guide*. This manual set the standard for future Commodore programmer's reference guides.

What Do You Do With A Computer?

The last obstacle to selling personal computers was that nobody knew what to do with them. This was the toughest challenge of all.

The key point in using a computer is that if you can find one useful, interesting, or practical application, you've justified its use. However, everyone has his/her own special need for a computer, and that's what makes this challenge so difficult.

One of the answers is to provide a useful selection of software. So, in addition to games, we introduced a *Home Calculation Six-Pack* and a *Personal Finance* program, and we developed some unique educational programs like the *Home Babysitter* cartridge, which contains three separate skill building programs for pre-schoolers.

We've also found that VIC owners are coming up with their own unique applications. A ninth grader wrote a program that keeps track of his paper route. A computer artist found a way to create new designs. A container executive uses the VIC to calculate complicated paper trim percentages.

After owning his VIC for two months, one VIC owner wrote a program and sold it to Commodore. And he'd never used a computer before.

A surprising number of people are taking up computing as a hobby, but they aren't really computer hobbyists, as hobbyists were defined a few years ago. These hobbyists aren't as technically inclined as those first hobbyists were in the 1970s. And they don't have to be technicians.

Remember the "chemistry set craze" in the 1950s and '60s? Nobody expected to invent medical cures or split atoms with their chemistry set, but they sure had a lot of fun with them. The same holds true for the VIC-20, except that a lot of people are finding that their experiments often lead to some practical or creative applications they can use in their home, school, or business.

Story Of The VICmodem

The device which lets you connect your computer to the telephone

is called a *modem*, but modems cost as much as \$400. In 1981 we wanted to develop a "VICmodem" which could retail for about \$100, but no one wanted to build a modem we could sell for that price. Everyone wanted to "protect" their price levels, or felt it wasn't "time" for a hundred dollar modem, or that it was technically impossible.

Finally, a small company that made industrial modems for food processing plants offered to help design our modem. After several tough sessions we came up with a modem on a cartridge which plugs directly into the VIC-20 and connects to any modular telephone handset. A Nonmodular telephone adapter for connecting the modem into the wall phone socket was also designed for those who didn't own modular telephones, and for users in Canada.

The final VICmodem includes a free subscription to and complimentary hour on CompuServe's information service (including "Commodore Information Network," a VICterm terminal program on tape) and several other special offers, all for only \$109.95. VICmodem went on sale only six months from the day the original idea popped into our heads.

VIC-20 As Home Appliance

Computers had a hard time being accepted as retail home appliances because of the chicken and the egg phenomenon. For example, the VIC-20 couldn't be accepted as a home appliance until a housewife or student could walk into a department store and buy the VIC-20 off the shelf, like a radio or an alarm clock. On the other hand, department stores weren't ready to put computers on their shelves until the general public was ready to come into their stores to buy computers off the shelf.

Commodore put together a consumer products team which went after the retail market and persuaded large department stores, toy stores, audio-video stores, and even discount chains to carry the VIC in large quantities.

We provided regional training for store personnel, designed an in-store display fixture containing a full selection of VIC products, and put together co-operative advertising and other merchandising programs which appealed to mass merchandising chains. The result is that the VIC-20 is now being sold in places like Macy's, Toys 'R Us, and even K mart, and is included on the back cover and inside Montgomery Ward's catalog.

What's next? There are some exciting surprises in VIC's future. And we will continue to produce new and better software for the

CHAPTER ONE

VIC-20, including more "practical" software which most serious computerists will appreciate.

The "story of the VIC" goes on.

Computer Genesis: From Sticks And Stones To VIC

DOROTHY KUNKIN HELLER / DAVID THORNBURG

A history of computing, from the fingers and bones of cave dwellers to the VIC.

In The Beginning:

The human race has been working on the invention of the computer for quite some time. Centuries ago, Neanderthal man counted on his fingers to provide simple calculations (the primeval digital system) and, in emergencies, on his toes. Neolithic cave dwellers scratched strange linear patterns on bones, which anthropologists now believe to be primitive calculators. The stones at Stonehenge may also have served as astronomical computers to indicate the rising and setting points of the sun, moon, and other planets.

From fingers and bones, rocks and stones, to UNIVAC, to Commodore's new sub-desktop personal computer is a long evolution – the end of a process of development that took hundreds of years – and, perhaps, the beginning of a new era in personal computing.

B.C.: Before Computer

Labor-saving devices to simplify the routine chores of "number-crunching" have preoccupied inventive minds throughout the ages. The abacus stands out as one of the more successful attempts, which was copied from the Chinese by the Greeks and Romans. The Western world contributed to the quest for automation by inventing the mechanical clock around 996 A.D., a device that calculates time by counting events. In the early 1400's, bankers and moneylenders used a simple, but effective method to count coins and compute interest – a checkerboard tablecloth (the original *check*).

In the 17th century, the process accelerated. Several kinds of calculators were developed in the 1600's that were capable of handling routine computations. In 1624, John Napier, a Scottish mathematician, invented logarithms – tables of numbers that greatly simplified multiplication and division. He then developed a set of

rotatable wooden cylinders for multiplication known as “Napier’s bones.” The “bones” could perform only trivial calculations, but were an economic success in that time of poor education. Related to the concept of logarithms, the slide rule was invented by an Englishman seven years later.

Meanwhile, a German inventor designed, but never actually built, the first mechanical calculator that would add, subtract, multiply, and divide.

It was Blaise Pascal, a French mathematical prodigy (and son of a tax collector), who constructed the first true calculating machine. The Pascaline was awkward by today’s standards, but was a very clever innovation for the 17th century. With cogs and wheels, it added, subtracted, multiplied, and divided and then displayed the result in a window.

However, no one bought it. Although the machine could make life easier for the underpaid clerks who performed the routine drudgery of 17th century accounting, these clerks wanted no part of the Pascaline, fearing that automation would cost them their livelihood. Employers anticipated that repair and maintenance on the Pascaline would be expensive, as was the initial investment. Since human labor was cheap and easily replaceable, the invention, although admired, remained unsold.

Leibnitz, another brilliant mathematician, toyed with calculators and the concept of binary numbers, but failed to marry the two. He invented a “multiplier wheel,” but became bored with it. He then became fascinated with the powerful potential of binary numbers, which make mathematical calculations possible using only two types of symbols: 0 and 1. (Eastern civilizations knew about binary notation centuries before the West, as shown in the *I Ching*.)

If Leibnitz had realized that the mass of complicated cogs and wheels necessary to run a machine like the Pascaline could be replaced with simple on-off levers, the age of B.C. might have ended in the middle of the 1600’s. Christopher Evans speculates, in his book *The Micro Millennium*, that giant, steam-driven computers in the 18th century could have been one result.

It was a French weaver from Lyons, Joseph Jacquard, who finally came closest to a functional contemporary computer – the first automated device that used punched cards to store a program and control a machine. This device is still in use today.

The “Programmed” Loom

Joseph Jacquard had every right to become discouraged at an early

age. His first invention, at age 16, was smashed to smithereens by his fellow workers and he was dismissed from his job. His device to manufacture knifeblades panicked the workmen who earned their living making them by hand. Undeterred, Jacquard made his life's goal the invention of an automated loom. He worked on this invention for 20 years, supported by his wife who worked in a millinery shop. Jacquard was partly inspired by the automated toys of an earlier French inventor who had made a mechanical duck that ate, quacked, waddled, and digested, as well as an automatic chess player and flute player.

Jacquard's career was almost cut off for the second time by the French Revolution. He first joined the anti-revolutionary forces and had to hide underground when the revolutionaries occupied Lyons. He then fled with his son, joined up with revolutionaries, and saw his son killed at his side. The Revolution and the ensuing unrest almost ruined Lyons, the center of the weaving industry in France. The weavers were saved only by government support to prevent France from losing the textile business entirely to England. In 1801, Jacquard won a government award of 20,000 francs for a machine that wove fishnets. Napoleon also gave Jacquard the job of repairing and arranging models and machines in the Conservatoire des Arts et Métiers.

There he found the missing Bouchon loom, an earlier invention that guided warp threads by means of holes punched in cards – a complex and expensive method of automating the weaving process. Jacquard developed a machine based on this loom that is still in use today. Horizontal steel rods with springs at the end “sense” the holes punched in a rectangular piece of cardboard. When a rod “feels” a hole, it passes through and activates a mechanism for lifting the appropriate warp thread, which is then skipped in the weaving, while other threads are regularly woven. The hole-punching *programs* the pattern.

Fearful of a return of the unemployment they had suffered during the French Revolution, mobs of working people attacked Jacquard. He attempted to reason:

My loom will save Lyonnaise industry. At present, only the rich can afford our fabrics. Tomorrow, by the grace of my machine, all classes of society will use them. What you will lose you will get back double. I shall bring about a prosperity to the manufacturers that will be a source of well-being to the workers.

The mob responded, “Yes, and while we wait we shall be

begging in the streets," and dunked him into the Rhone River.

Several more attempts were made on Jacquard's life by threatened working people.

The loom was finally used because of foreign competition; by 1812, 11,000 automated looms were in operation in France.

Jacquard was honored in 1819 by the Chevalier d'Honneur. In 1834, the English inventor, Charles Babbage, saw a portrait of Jacquard done in silk thread by a Jacquard loom with 24,000 cards, each punched with 1,050 holes. This inspired Babbage with the idea of "programming" his Analytical Engine with punched cards.

The Weaver Of Numbers

Charles Babbage spent his long life and considerable fortune in his quest to develop the Analytical Engine, a machine that could evaluate any mathematical formula and was designed to have most of the capabilities of a contemporary computer.

Babbage first attempted a "Difference Engine" that would solve polynomial equations. He received the Royal Astronomical Society's first gold award, in 1822, for his paper on "Observations on the Application of Machinery to the Computation of Mathematical Tables," and received grants from the British government for development. Many years and thousands of pounds later, the machine was still not functional because of mechanical difficulties. Slight imperfections in the hundreds of rods, wheels, ratchets, and gears that made up the complicated machine's working parts caused the whole system to break down or become inaccurate. In 1833, the British government gave up on the project. Characteristically, Babbage responded by proposing an even more ambitious project.

He proposed to construct a machine that had no fixed purpose, but could handle a variety of tasks based on the owner's instructions. The task was how to tell the machine what to do: to set up a unique sequence of internal activity, with each different task being tied to unique "patterns of action." In other words, a programmable computer.

His actual machine never really worked, but it had all of a computer's component parts: input devices, a processor, a control unit, a memory, and an output mechanism – all mechanical, of course.

Babbage was aided by a synergistic partnership with a remarkable woman, Ada, Countess of Lovelace, the first systems programmer in history. She was the daughter of the Romantic poet, Lord Byron, and an amateur mathematician who was known as "the Princess of

Parallelograms" in her youth. Ada was young, brilliant, but frustrated by her domineering mother and constricted Victorian home life.

Babbage seems to have been a "right-brain," whimsical, inventive person. Ada was a "left-brain" person with (in her own words) "a clear, logical and accurate mind." They complemented each other perfectly.

Ada's copious notes on Babbage's invention, according to a 20th-century mathematician, "show her to have fully understood the principles of a programmed computer a century before its time." Ada translated and annotated a paper on the Analytical Engine written by an Italian engineer, L. F. Menabrea. While doing so, she discovered serious errors in Babbage's work. She also planned many problems (or programs) for the Engine to carry out; described plans to use punched cards for data input and output and to program the machine; and suggested a binary system of storage instead of the decimal system Babbage was using.

Together, they started the continuing debate on the intellectual capacities of computers: can a machine think? Babbage personified the workings of his machine, and even wrote of its "feelings." Ada, a real systems programmer, unequivocally stated: "The Analytical Engine has no pretensions *to originate* anything. It can do whatever *we know how to order it* to perform. It can *follow* analyses; but it has no power of anticipating any analytical relations or truths. Its purpose is to assist us in making *available* what we are already acquainted with."

Not lacking in poetry or imagination, she envisioned a future where machines could play music and beat human opponents at tic-tac-toe. She also described the Analytical Engine as a weaver of numbers: "We may see most aptly that the Analytical Engine weaves algebraical patterns just as the Jacquard loom weaves flowers and leaves."

Ada died in agony of cancer at age 36, after a life of Victorian conflicts and scandals. Babbage lived on to see a Swedish engineer receive acclaim for producing a working model of the Analytical Engine, a project that Babbage himself was incapable of completing. He died a disappointed man, but not without making this prophetic comment: "As soon as an Analytical Engine exists, it will necessarily guide the future course of science."

When 10 = 2, It's A Computer

Babbage's invention was followed by a complex analog computer called the "Harmonic Analyser" by another British inventor, Lord

CHAPTER ONE

Kelvin. The Harmonic Analyser was capable of predicting tides years in advance; and Kelvin produced an improved model in 1876.

In the United States, a much more important event in the annals of computerdom was taking place. Herman Hollerith and John Shaw Billings conceived a punched card system for processing census data. Their system successfully tabulated the 1890 census in six weeks. What was even more significant for the future was the company they formed, now known as IBM.

The punched card method was a successful attempt to automate a process, but it wasn't a computer. Babbage and Lady Lovelace had provided clear specifications for a general-purpose computer, but the technology to build one that wasn't unwieldy, expensive, unreliable, and slow didn't yet exist.

In the 1930's, scientists and engineers in several countries began working towards the technological advances that would make the computer a practical reality. In the United States, IBM and the Bell Telephone Company were working on the problems of computerization. Alan Turing, an English mathematician, published his paper "On Computable Numbers," generally agreed to be the single most important paper in the history of computer science (by those capable of understanding it). It was a rigorous mathematical demonstration that a general-purpose computer such as Babbage had envisioned could be constructed. Turing was also a seminal figure in the development of "Colossus," the British codebreaking machine that may have enabled the Allies to win World War II.

In Germany, a young graduate student skilled in mechanics and engineering, but with little background in advanced mathematics or electrical engineering, constructed the first automatic programmable calculating machine. Konrad Zuse realized, in 1936, that the binary system was best suited for automatic calculating machines and designed a method of representing binary digits through a system of mechanical relays. His abstract representation of binary code involved three switches: AND, OR and NOT.

Zuse's digital device resulted in the Z-1 computer. His Z-2 featured electromagnetic circuits; his Z-3, an all-relay device. A friend and colleague, Helmut Schreyer, suggested that Zuse use vacuum tubes and later wrote a thesis on the subject that became lost in the archives, largely unread. Had the Nazi military machine connected the capabilities of the Z-3 with Schreyer's inspired suggestion, this might have enabled Germany to win World War II. Zuse evacuated his Z-4 to Switzerland at the end of the war and then concentrated on theoretical work. His "Plan Kalkul" system of notation contains

many of the features of the higher level languages (BASIC, etc.) used today. Zuse also designed a hard-wired compiler and founded a company that was later absorbed by Siemens.

While Zuse was working in Germany, a general-purpose computer was being built at the University of Pennsylvania. ENIAC (Electronic Numerical Integrator and Computer) contained over 18,000 vacuum tubes and was hundreds of times faster than previous machines. However, it worked on a decimal rather than a binary system. It was massive and had a tiny memory. It was first programmed by unplugging and rearranging pathrods, and had to be rewired each time a program was changed. Fortunately, a member of the ENIAC team happened to start a conversation with John Von Neumann, a world-famous mathematician, while both were waiting for a train. Von Neumann's contributions to the ENIAC project resulted in the development of the stored program, which, according to Christopher Evans, "is the major single factor which allowed computers to advance way beyond the power of ENIAC and its various contemporaries – a concept of fundamental importance."

The First Mainframe

The first of the mainframes was designed and constructed for the U.S. Government by two key members of the ENIAC team – Presper Eckert and John Mauchly. Said Eckert: "I had a skill in electronics, some knowledge of math, some knowledge of computers. John had greater skill in math and desk calculators... Our efforts synergistically allowed us to develop the first computer."

Univac I, the first commercial computer, was delivered to the U.S. Census Bureau in 1951, the first electronic computer purchased by any government agency. (A. C. Nielsen & Co. almost bought UNIVAC to process opinion poll data, but backed out.)

The UNIVAC team included another distinguished programmer who is still active in the field and holds the title of Commander in the U.S. Navy. Dr. Grace Murray Hopper, one of the early developers of COBOL, made an observation on programmers which led to the development of the first compiler. The UNIVAC programming team often had to copy code from each other's notebooks. "One of our startling discoveries," she recalled, "was that programmers cannot copy things and programmers cannot add."

Although UNIVAC was a major breakthrough, its capacity and speed were minimal by today's standards, and its physical size was alarming – part of a government building had to be knocked down to move it. However, UNIVAC's developers were major tradition-

breakers. "John (Mauchly) saw things really as what they were, not what people told him they were," said Eckert. At a recent UNIVAC reunion, Grace Murray Hopper encouraged continued innovation: "There is one great danger in computing today," she said, "and that is that phrase 'But we've always done it that way.' "

From The Garages Of Silicon Valley

"Who could have imagined that the development of the integrated circuit would follow so soon after the invention of the computer?" remarked Eckert in 1980.

In 1965, the integrated circuit – many components on a single piece of semiconductor – was introduced. In 1971, INTEL introduced the first microprocessor chip, the 4004. The microprocessor was designed to eliminate random logic in complex digital circuits, the goal being to reduce chip count. None of the manufacturers realized at the time that the tiny, but potent, chip would open the door to a whole new field.

The 8080 eight-bit microprocessor was introduced by INTEL in 1974, as was the 6502 microprocessor chip by MOS Technology. In the garages of "Silicon Valley," California, a small area of the southern San Francisco Peninsula where high-technology companies were beginning to replace the prune and cherry orchards, the computer revolution had begun. In one home hobbyist workshop, Kentucky Fried Computer was born. After receiving an indignant letter from the Colonel, the new company changed its name to North Star. Also in 1974, MITS advertised computer kits built around the INTEL 8080A. MITS ran out of stock after the first ad. Expecting to sell 800 kits in 1975, they sold 2,000 and could not manufacture enough kits to keep up with the demand. In 1975, the first production microcomputer appeared – the Altair 880. Another computer company was born in this year, but not in a garage. Lore Harp and her friend, Carole Ely, decided to go into business with Lore's husband. Bob Harp, an electronic engineer, designed a memory board that would fit into other people's microcomputers and the team set up a workshop in Ms. Ely's bathroom. They shipped 4,000 boards in 12 months. Their company is now Vector Graphics; the two women became President and Secretary/Treasurer.

The first personal computers appeared in 1977 – the Apple I from Apple Computer Company, Radio Shack's TRS-80, and Commodore's PET.

The PET, according to one commentator, "took the world by storm." In addition to its technical features, it was produced by the

first vertically integrated computer company. Commodore International also owns an IC house, MOS Technology, whose 6502 chips appear inside almost all competitors. By 1978, Commodore had sold 25,000 machines. The PET was even more successful in Europe and still maintains the largest installed user base of any personal computer in England and Germany. Commodore then introduced the CBM (Commodore Business Machine) for business applications.

By 1981, the 250,000th microcomputer was sold in the United States – a remarkable growth from zero, to hobbyist kits, to all-purpose personal computer.

A New Era In Personal Computing

Some observers contend, however, that the computer revolution hasn't really begun. Current microcomputers, they claim, are solely for businesspeople or hobbyists. The "technoid" mentality of the technological/hobbyist elite, who still control the computer establishment, limits the personal computer as a real tool for the general public. Michael Tomczyk of Commodore stated the problem in 1980 in *Retail Computing*: the average (non-technical) consumer is intimidated by the computer, and the technical jargon and aura that surround it, and is unclear about the uses that would justify a sales price of several thousand dollars. Microwave ovens heat food faster, television (theoretically) entertains, washing machines wash: but what is a "personal computer" for?

To bridge the gap between the "technoid" personal computer and the mass electronic tool of the future, Commodore introduced the VIC-20 sub-desktop model in March 1981. In advance of the anticipated "Japanese invasion" of the U.S. computer industry, Commodore is manufacturing the VIC in Japan and marketing it here as a direct competitor to Japanese computers on their home ground.

What is more significant to the thousands of new computer users who are buying the VIC-20 is that it is easily accessible to them both economically and technically – perhaps the beginning of a new era in personal computing.

Super Calculator

JIM BUTTERFIELD

VIC has special functions similar to an advanced scientific calculator that you can easily use from the keyboard.

Everyone knows that you can load programs into the VIC and get some pretty clever things to happen. Don't forget that your VIC can also do useful tasks without any programs at all.

The technique is called Direct Statements. These are lines that you type without a number at the beginning. For example, if you type `PRINT "HELLO"` or just `? "HELLO"` for short, VIC will obligingly print HELLO. Not too useful, but we're just warming up for the good stuff.

Quick arithmetic is easy to do. To add five and six, type `PRINT 5 + 6`. It works just as you expect it to.

VIC uses an `*` (asterisk) character to signify multiplication, and a `/` (slash) for division. So `PRINT 2*3/4` gives you an answer of 1.5 as you would think. By the way, you'll quickly learn that VIC ignores spaces: `PRINT 2 * 3 / 4` gives the same result, and you may feel that it's neater.

When you start mixing multiplication/division with addition and subtraction, you'll need to get used to a quick VIC trick: it always performs the multiplication and division first. This means that `PRINT 2*3 + 4*5` will produce 26 (6 plus 20), not 50 as you might think at first. If you really wanted to multiply by five you could always force VIC to see things your way by using brackets: `PRINT (2*3 + 4)*5` makes it work. You can use multiple brackets if you wish: `PRINT ((2 + 3)*4) + 5` is quite acceptable, and `PRINT (2 + 3)*(4 + 5)` produces the expected answer of five times nine or 45. Remember that you must close the brackets as many times as you open them, or you may get the dreaded `?SYNTAX ERROR` notice that tells you you've done something dumb. If you'd rather be exact about brackets and call them parentheses, that's OK – just remember to use them correctly.

You'll quickly discover that you can raise a number to a power with the upward arrow: `PRINT 2 ^ 3` gives you two cubed, which is eight. Powers are always performed before multiplication, division,

addition or subtraction – unless you use brackets. By the way, you'll discover that powers of a number have one very nice feature: the sign of a number is handled correctly in almost all cases. If you have a mathematical bent, you can probably guess what will happen if you raise a number to a fractional or negative power; if you don't, you might like to try it anyway and see what happens. One last thing about powers: they don't work out exactly in all cases: three raised to the fourth power might give you a value just a shade higher or lower than 81.

We've still only just begun. VIC has special functions similar to an advanced scientific calculator. For example, `PRINT SQR(5)` calculates and prints the square root of five. You have quite a few trigonometric functions: `SIN`, `COS`, `TAN` and the arctangent `ATN` if you need them, but be careful: they are worked from angles in radians. If you measure your angles in degrees, be sure to convert using a factor of $\pi/180$: for example, the sine of 30 degrees is calculated with: `PRINT SIN(30 * π / 180)`. For the math whiz, there are logarithms and exponentials using the `LOG` and `EXP` functions. If you use these, you'll need to know that they are natural logarithms. If you prefer to use unnatural logarithms (base 10), use a factor of `LOG(10)` to divide or multiply: the common log of two can be calculated with `PRINT LOG(2)/LOG(10)`.

Memories

Calculators use memories, and VIC, the super-calculator, gives you lots of memory. You get to name your memory: type `A = 17` and the value of 17 is stored into a memory location called A. Later, you can use this value in other calculations such as `PRINT A + 9`. You can change the memory value at any time with a statement like `A = 14`. You can add or subtract to it with unusual (at first) syntax such as `A = A + 4` or `A = A - 11`. When you do this kind of thing, remember that the new value is set only after the calculation is complete. So if A equals 5, the expression `A = A * 3 - A` would calculate five times three minus five, and then set the result (ten) into memory location A.

You may name memory locations (called "variables" in the VIC) almost anything you like: for example, `HENRY = 7` will work. You'll be much better off to use a single letter (A, B, C, etc.) or a letter followed by a number (D9, M4, etc.) since VIC can get confused with certain combinations of letters. For example, `TANK` would get mixed up with the `TAN` function.

One last thing: memory can get wiped out very easily in the VIC. Certain commands like `NEW` and `CLR` will do it; and typing

in any line starting with a line number will clear all the variables. Be careful.

Multiple Calculations

If you want to calculate several things, you can do it with a single PRINT command. Just put a semicolon (;) or a comma (,) between the expressions you want to calculate. For example: PRINT 3 + 5, 3*5, 3/5 calculates three values and prints them neatly on a single line. If you used the semicolon: PRINT 3 + 5; 3*5; 3/5 the values would be printed close together rather than in neat columns.

You can put several commands together on a single line, separated by a colon (:) character. To add a value of one to variable X, and then print X + 4, you might type, X = X + 1 : PRINT X + 4. The colon will separate the statements so that VIC will understand that they are to be performed separately.

Repeating Calculations

If you can ask VIC to do something once, you can ask for the same calculation to be performed many times. All you need to do is put the job you want done between the two following statements: FOR J = 1 TO ... : ... (your statement) ... : NEXT J.

Beginners like to see their name printed many times. They should code: FOR J = 1 TO 100 : PRINT "JOE":NEXT J to have the name JOE printed one hundred times. Since each name is printed on a separate line of the screen, there won't be room for all those JOEs. Try changing the PRINT statement by adding a little extra punctuation after JOE – for example, PRINT "JOE", with a comma, or PRINT "JOE"; with a semicolon. (Make sure the punctuation is outside the quotation marks.) You have a lot of control over how things appear on the screen.

It doesn't seem to make much sense to print a fixed calculation such as the square root of ten over and over again. We have much more flexibility than that. As the central statement repeats, the value of the variable (J in the example above) will step through the values we have shown (1 to 100). You may use this variable as a memory value and calculate with it. To print a table of the square roots of numbers from ten to twenty, code: FOR J = 10 TO 20 :PRINT J,SQR(J) :NEXT J and the job will be done.

Remember that everything between the FOR and NEXT statements will be repeated with the value of the variable (J in this case) stepping through its range. Don't forget that you can use any variable name you like: FOR M = 3 TO 7 is perfectly good so long as you say NEXT M at the point that you want to go back and repeat.

Using Direct Statements

Direct statements are a good way to learn some of the simple rules of BASIC, and they are handy for quick calculations, too.

When you start writing BASIC programs, you'll find it handy to try some of the program lines as direct statements first, to make sure that they work properly. And if your program gives you trouble and stops, you'll find that statements from the program, entered in Direct mode, can give you a hint as to what's going on.

But no matter how advanced you get in your programming adventures, don't forget what a zippy little calculator you have at your fingertips.

Large Alphabet

DOUG FERGUSON

Using custom characters and a special feature of the VIC, you can program large, double-height characters.

There are many exciting applications for the 64 programmable characters on the VIC-20. David Malmberg's "Custom Characters" [*reprinted in this book*] explains fully how the VIC can generate programmable characters merely by changing the contents of memory location 36869 (decimal), and by redefining the 64 eight-pixel tall characters beginning at 7168.

Another interesting memory location in the VIC is nearby: 36867. Changing its contents creates double-sized characters. By POKEing a 47 into 36867, the bottom border of the screen drops out of sight and vertically paired characters occupy "stretched" screen locations. After clearing the screen, type an A and get $\frac{B}{2}$. Actually, the VIC's first character is the "@" (screen POKE 0) which yields $\frac{A}{\lambda}$. Continue to type the alphabet and see how the stacked letters follow a pattern. To return to normal, POKE 36867,46 or hit the RESTORE and RUN/STOP keys simultaneously.

I set about to combine these two ideas so that I could get a large alphabet. I painstakingly reprogrammed the B to look like the top of a stretched "A" and the C to look like its matching bottom half. Continuing for nearly two hours, I made it to the "O" and gave up for the night.

Somehow, the clear light of day the next morning directed me toward a much simpler approach: if the characters already reside in ROM, just read each eighth of a character *twice* into the RAM space for programmable characters to program two letters at a time!

Clearly, only 32 such stretched characters can be made since only 64 unstretched characters can be readily programmed. The space key and all the numerals fall in the wrong half of the 64, but all 26 letters of the alphabet can be stretched with the following, suprisingly short, program:

```
10 POKE 56,28 : REM RELOCATE END-OF-MEMORY PO
  INTER
20 CH=32776    : REM LOCATION OF ALPHABET IN R
  OM
```



```
30 FOR X=7184 TO 7600 STEP 2 : REM ALPHABET I
  N RAM
40 POKE X,PEEK(CH):POKE X+1,PEEK(CH):REM STRE
  TCH
50 CH=CH+1:NEXT X : REM LOOP
60 POKE 36879,25 : REM NO MORE BORDER
70 POKE 36869,255 : REM PROGRAMMABLE CHARACTE
  RS
80 POKE 36867,47 : REM STRETCHED CHARACTERS
90 PRINT"{CLEAR}ABCDEFGHIJKLMNOPQRSTUVWXYZ":E
  ND
```

Lines 20 through 50 read the normal alphabet (8x8 pixels) out of ROM and into RAM. Since RAM is also where a longer program will do its work, line 10 tells the computer not to go beyond 7168 (28 times 256). Line 60 is for the purist who notices the lack of a bottom border with the "normal" screen.

Simple? Certainly. The biggest drawback is the lack of numerals and spaces. In string variables with spaces, e.g., `AS="HELLO THERE"`, the space can be replaced by the symbol for cursor-right.

The applications of this large alphabet program are left to the reader. Although it is obvious that any characters can be programmed for stretching, only the alphabet (and a few insignificant symbols) can be programmed in a way that an exact keyboard-to-character correspondence can be realized.

Using A Joystick

DAVID MALMBERG

A collection of programming hints, with two games to illustrate them – “Sketch” and “Spacewar.” Temporarily changing line 140 of Spacewar to 140 POKE52,28:POKE 56,28:CLR will allow you to make changes and RUN the program without running out of memory.

The designers of the VIC showed forethought and unselfishness when they designed it to use widely available Atari peripherals. Specifically, the Atari light pen, game paddles, and joystick can be plugged into the VIC game port and used to create some very exciting programs. However, the mechanics of just how to use these peripherals in a VIC program have not yet been documented by Commodore in the United States.

After briefly describing how joysticks work in general, we'll discuss a program which allows you to draw either lines or graphic characters in any color on the VIC screen, using the joystick to control the direction of the drawing. Finally, this article presents an exciting and action-packed version of the arcade game “Spacewar” in which joystick control is critical to making the game easy and fun to play. After studying these two programs, you should be a joystick “expert” and should be able to easily incorporate the joystick into your own VIC programs.

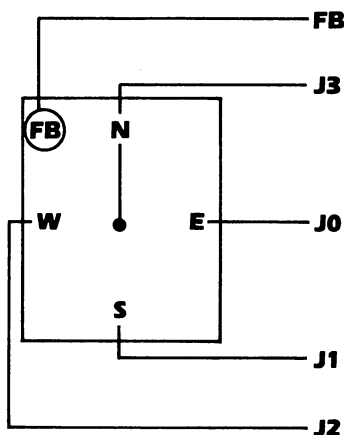
How Joysticks Work

All joysticks work using one of two basic approaches. The first approach has the relative position of the joystick control two voltages – one corresponding to the horizontal displacement of the joystick and the other corresponding to the vertical displacement. This is done by having the joystick attached to two potentiometers whose resistances are determined by the position of the joystick. The output voltages are then read into a device that can convert the analog voltage to a digital value the computer can understand. Using this approach, it is possible to have the joystick determine a specific position on the screen. For example, if the x-voltage (horizontal) and y-voltage (vertical) both ranged from zero to five volts, an x reading of zero volts and a y reading of 2.5 volts would correspond to the first column and middle row of the screen. The VIC does not use this

technique for the Atari joystick, but it does use this approach for reading the game paddles – so it would be possible to set up this type of joystick for the VIC by reading the x and y voltages as if they were different game paddles.

In contrast to reading a specific screen position by its x and y coordinates, a second approach reads the joystick's direction. This is the design philosophy used in the Atari joystick. It does this by reading a series of switches corresponding to the basic compass directions as shown in the following diagram:

Figure 1.



For example, if the joystick is pushed north (up or forward) as shown in the diagram, switch J3 will be closed and all other switches will be open. If the joystick is pushed on the diagonal, the two nearest switches will be closed simultaneously; e.g., J1 and J2 will both be closed if the joystick is pushed to the southwest. Notice that these switches imply only the direction of the joystick and that it is up to the program to keep track of any corresponding screen position or movement.

To see how to make the VIC read these switches, plug in your joystick and enter and run the following short program:

```
10 DD=37154:P1=37151:P2=37152
20 PRINT"{CLEAR}{DOWN} N E S W FB"
30 GOSUB 100:PRINT"{HOME} ";J3;J0;J1;J2;FB
40 GOTO 30
100 REM SUBROUTINE TO READ JOY SWITCHES
110 POKE DD,127:P=PEEK(P2)AND128
```

CHAPTER ONE

```
120 J0-- (P=0)
130 POKE DD,255:P=PEEK(P1)
140 J1-- ( (P AND 8)=0)
150 J2-- ( (P AND 16)=0)
160 J3-- ( (P AND 4)=0)
170 FB-- ( (P AND 32)=0)
180 RETURN
```

As you run the program you will notice that, whenever you push the joystick in a particular direction, the corresponding compass direction(s) will change from a zero to a one. Similarly, whenever you push the fire-button, it changes to a one – no matter which direction the joystick is being pushed at the time.

The reason why and how this short program actually works is beyond both the scope of this article and the interest level of most readers. I will leave it to Commodore to explain more fully whenever they issue their documentation on the joystick. Suffice it to say, it does work!

Make A Sketch

Program 1 is a simple program that illustrates the use of the VIC joystick to draw either lines or graphic characters on the screen in any color. While the program is running, you may control your artistic endeavors by:










- Hitting an "L" to change from graphics to lines
- Hitting a "C" to start all over again with new options
- Pressing the fire-button to clear the screen
- Hitting any color key to change color
- Hitting any other key to change to its graphic character

Let's review this program line by line to learn how to use the joystick effectively. Lines 140 and 150 define some of the important VIC locations and parameters as variables. Not only does this save memory by requiring fewer bytes to reference these values later, but it is also faster because the VIC can look up a variable in its variable table much quicker than it can decode that same value expressed as a constant. C is the number of columns on the VIC screen; i.e., 22. R is the number of rows. S is the starting location of screen memory. A is the constant that must be added to a particular screen memory location to get its corresponding color matrix location; i.e., $S + A$ is the start of the color matrix. DD, P1 and P2 are the same values as used in the short program above to allow us to read the joystick. SB

is the location that is POKEd to set the screen and border color combinations. V is the volume location for the sound and S1 is one of the sound "voices."

The values in the matrix L%(I,J), as defined in lines 160 and 170, contain the values that, when POKEd to the screen, draw a line at a certain angle. Specifically, L%(I,J) contains these screen poke characters:

Figure 2.

		I		
		0	1	2
J	0			
	1			
	2			

These values will be used later when drawing lines on the screen.

Line 180 defines two very useful functions. FNA calculates the screen memory location corresponding to any row (Y-value) and any column (X-value). FNC is the corresponding color matrix location.

Lines 190 to 360 give instructions (if desired) and then ask for the initial values of the various user options, such as color, starting drawing location, character, speed, etc. Line 370 sets the screen and border colors to whatever combination has been specified and then also clears the screen.

Line 390 is the beginning of the main processing loop and GETs any key that is hit. If a key has been hit, lines 400 to 440 change to drawing lines, change color, clear the screen and start over, or change the graphic character being drawn – whatever is appropriate. If no key were hit, the program would jump to line 450 which is a GOSUB680. This subroutine is essentially the same one used in the short program above to read the joystick switches.

After returning from the joystick subroutine, lines 460 to 510 determine the current relative directions, DX and DY, and the new screen coordinates, X and Y. DX is -1, 0, +1 if the joystick is left, center, or right, respectively. Similarly, DY is -1, 0, or +1 if the joystick

CHAPTER ONE

is up, center, or down. Lines 530 to 560 check and restrict the values of X and Y to values within the screen border.

If the program is drawing lines (i.e., the line flag L has been set to 1), line 570 sets the current poke character, CR, to the appropriate matrix value of $L\%(DX + 1, DY + 1)$. For example, if the joystick is being pushed northeast, DX will be +1, and DY will be -1, and the value of $L\%(2,0)$ is a poke character of a line going up and to the right.

Lines 580 and 590 POKE the reverse of the current character ($CR + 128$) at the current screen coordinates, POKE the current color into the corresponding color matrix location, and sound a hi-note for DL jiffies (i.e., a 60th of a second). Lines 610 to 630 POKE the current character (not reversed) at the current screen coordinates, sound a lo-note for another DL jiffies, then turn off the sound. Line 640 loops back to the beginning of the whole process at line 390.

Spacewar

Programs 2 and 3 contain a version of the classic arcade game "Spacewar" which absolutely requires a joystick in order to maintain both the speed and sense of excitement of the original game. The game is in "hi-res" using the VIC's custom character feature (*Home and Educational COMPUTING!*, Summer 1981) and is a reasonably close copy of the original game.

In the game, you command a single spaceship (using the joystick, of course) against a kill-crazed, kamikaze Mingon. There are nine levels of play, ranging from trivial to impossible. You must shoot him (using the fire-button) before he gets you. There are asteroids, black holes, and the sun's gravity to add complications. If you get desperate, you can try a hyperspace jump – but there is no guarantee you will survive.

I will not attempt to describe "Spacewar" on a line-by-line basis. But, as you key it in, you will recognize most of the joystick techniques used in "make a sketch." The code is difficult to follow because of the short variable names and the dearth of REMs – all necessary to enable it to fit in the VIC's 3.5K of memory.

I used one other memory-saving trick that might be of interest. The program is actually written in two separate parts. Program 2 gives instructions, asks for the various game options, loads the custom character set into hi-memory, and then resets the memory pointers to protect the special character set. Then this part of the program automatically loads and begins execution of the second part, Program 3, which is the actual game. This trick is done in line

2190 of Program 2 which POKes the keyboard buffer with the commands NEW, LOAD and RUN. When the VIC encounters the END at the end of line 2190, it quits executing Program 2 and then it checks the keyboard buffer for additional commands, which it executes as if they were keyed in directly by you. This enables 7K worth of programming to appear to fit into a 3.5K VIC. You should keep this trick in mind whenever you have a program that just will not fit in the VIC's limited memory.

After studying these two programs you should be a joystick master and be able to easily incorporate the joystick into your own VIC programs. This will make your programs both more interesting and more exciting.

Program 1. Make a Sketch.

```

100 REM VIC MAKE-A-SKETCH USING JOYSTICK
110 REM BY DAVID MALMBERG
120 REM 43064 VIA MORAGA
130 REM FREMONT, CALIFORNIA 94538
140 C=22 : R=23 : S=7680 : A =30720 : DD=
    37154 : P1=37151 : P2=37152
150 SB=36879 : V=36878 : S1=V-2 : DIML%(2
    ,2)
160 FOR I=0TO2 : FOR J=0TO2 : READ L%(I,J
    ) : NEXT J,I
170 DATA 77,64,78,93,46,93,78,64,77
180 DEF FNA(Z) = S+(X-1)+C*(Y-1) : DEF FN
    C(Z) = FNA(Z) + A
190 PRINT"{CLEAR}          {REV}MAKE-A-SKETCH{O
    FF}" : INPUT"{02 DOWN}INSTRUCTI
    ONS  N{03 LEFT}";A$
200 IF A$="Y" THEN GOSUB720
210 POKESB,27 : PRINT"{CLEAR}{REV}MAKE-A-
    SKETCH  OPTIONS{OFF}"
220 PRINT"{02 DOWN}ENTER BORDER COLOR" : ~
    GOSUB650
230 BR = VAL(A$)-1 : IF BR<0 OR BR>8 THEN
    220
240 PRINT"{DOWN}ENTER SCREEN COLOR" : GOS
    UB650
250 BC = VAL(A$)-1 : IF BC<0 OR BC>8 THEN

```

CHAPTER ONE

```
240
260 PRINT"{DOWN}DRAWING SPEED ?" : PRINT"
{REV}0{OFF}-FAST TO {REV}9{OFF}-
SLOW" : GOSUB650
270 DL=VAL(A$)
280 PRINT"{DOWN}START IN CENTER ?" : GOSU
B650
290 IF A$="Y" THEN X=10 : Y=11 : GOTO340
300 INPUT"{DOWN}STARTING ROW 8{03 LEFT}"
;Y
310 IF Y>22 OR Y<0 THEN 300
320 INPUT"{DOWN}STARTING COLUMN 8{03 LEF
LEFT}";X
330 IF X>23 OR X<0 THEN 320
340 PRINT"{DOWN}STARTING CHARACTER ?":PRI
NT"{REV}L{OFF} WILL GIVE LINES":
GOSUB650
350 CR=ASC(A$) : L=0 : IF A$="L" THEN L=1
360 PRINT"{DOWN}STARTING COLOR ?":GOSUB65
0:CL=VAL(A$)-1:IF CL<0 OR CL>8 T
HEN 360
370 POKESB,16*BC+BR+8 : PRINT"{CLEAR}"
380 REM BEGINNING OF DRAWING LOOP
390 A$="" : GETA$ : IF A$="" THEN 450
400 IF A$="L" THEN L=1 : GOTO450 : REM US
ES LINES
410 IF A$>"0" AND A$<"9" THEN CL=VAL(A$)-
1 : GOTO450 : REM NEW COLOR
420 IF A$="C" THEN 210 : REM START OVER
430 NC=ASC(A$) : IF NC>127 THEN 390 : REM
INVALID CHARACTER
440 CR=NC : L=0 : REM CHANGE CHARACTER
450 GOSUB680 : REM READ JOYSTICK SWITCHES
460 DX=0 : DY=0 : IF J0 THEN DX=1
470 IF J1 THEN DY=1
480 IF J2 THEN DX=-1
490 IF J3 THEN DY=-1
500 IF FB THEN 370 : REM FIRE BUTTON CLEA
RS SCREEN
510 X=X+DX : Y=Y+DY : REM NEXT COORDINATE
S
```

```

520 REM TEST AND ADJUST FOR SCREEN BORDER
530 IF X<1 THEN X=1
540 IF X>C THEN X=C
550 IF Y<1 THEN Y=1
560 IF Y>R THEN Y=R
570 IF L THEN CR=L%(DX+1,DY+1) : REM LINE
    CHARACTER
580 TT=TI : POKE FNA(0),CR+128 : POKE FNC
    (0),CL : REM POKE REVERSE
590 POKE V,15 : POKE S1,225 : REM HI-SOUN
    D
600 IF TI-TT<DL THEN 600 : REM SPEED DELA
    Y
610 TT=TI : POKE FNA(0),CR : POKE S1,135 ~
    : REM NORMAL CHARACTER
620 IF TI-TT<DL THEN 620 : REM ANOTHER DE
    LAY
630 POKE V,0 : POKE S1,0 : REM SOUND OFF
640 GOTO390 : REM END OF LOOP
650 A$="" : GETA$ : IF A$="" THEN 650
660 RETURN
670 REM SUBROUTINE TO READ JOYSTICK SWITC
    HES
680 POKE DD,127 : P=PEEK(P2)AND128 : J0=-
    (P=0)
690 POKE DD,255 : P=PEEK(P1) : J1=-((PAND
    8)=0)
700 J2=-((PAND16)=0) : J3=-((PAND4)=0)
710 FB=-((PAND32)=0) : RETURN
720 PRINT"{CLEAR}{REV}MAKE-A-SKETCH{OFF} ~
    ENABLES"
730 PRINT"YOU TO DRAW LINES OR"
740 PRINT"GRAPHIC CHARACTERS ON"
750 PRINT"THE VIC SCREEN USING"
760 PRINT"A JOYSTICK."
770 PRINT"WHILE YOU ARE DRAWING"
780 PRINT"YOU MAY CONTROL THE"
790 PRINT"ART IN THESE WAYS:"
800 PRINT"  Q - HITTING AN 'L'"
810 PRINT"CHANGES TO LINES"
820 PRINT"  Q - HITTING A 'C'"

```

CHAPTER ONE

```
830 PRINT"STARTS ALL OVER"
840 PRINT" Q - PRESSING THE"
850 PRINT"FIRE-BUTTON CLEARS"
860 PRINT"THE SCREEN"
870 PRINT" Q - HITTING A COLOR"
880 PRINT"KEY CHANGES COLOR"
890 PRINT" Q - ANY OTHER KEY"
900 PRINT"WILL CHANGE THE"
910 PRINT"CHARACTER BEING DRAWN"
920 PRINT"{DOWN}PRESS {REV}RETURN{OFF}" :
    GOSUB650
930 RETURN
```

Program 2. Spacewar Part 1.

```
100 POKE36879,27:PRINT"{CLEAR}          VIC SP
    ACEWAR"
110 PRINT"{02 DOWN}  BY DAVID MALMBERG"
120 REM 43064 VIA MORAGA
130 REM FREMONT, CALIFORNIA
140 X=PEEK(56)-2:POKE52,X:POKE56,X:POKE51
    ,PEEK(55):CLR
150 CS=256*PEEK(52)+PEEK(51):S=7680
160 FORI=CSTOCS+511:POKEI,PEEK(I+32768-CS
    ):NEXT
170 READ X:IFX<0THEN200
180 FORI=X TO X+7:READJ:POKEI,J:NEXT
190 GOTO170
200 INPUT"{02 DOWN}    INSTRUCTIONS  N{03
    LEFT}";A$
210 IFA$="N"THEN2000
220 PRINT"{CLEAR}YOU ARE ENGAGED IN A"
230 PRINT"BATTLE TO THE DEATH"
240 PRINT"WITH A KILL-CRAZED,"
250 PRINT"KAMIKAZE MINGON IN"
260 PRINT"THE ASTEROID BELT OF"
270 PRINT"MONGO."
280 PRINT"{DOWN}YOUR SHIP IS PURPLE"
290 PRINT"AND THE MINGON SHIP"
300 PRINT"IS DARK BLUE."
310 PRINT"{DOWN}THE JOYSTICK CONTROLS"
```



```

320 PRINT"YOUR DIRECTION.  THE"
330 PRINT"RED BUTTON FIRES YOUR"
340 PRINT"LASER MISSILES.  MAKE"
350 PRINT"A HYPERSPACE JUMP BY"
360 PRINT"PRESSING THE SPACE"
380 PRINT"BAR."
390 PRINT"{DOWN}PRESS ANY KEY"
400 A$="":GETA$:IFA$=""THEN400
410 PRINT"{CLEAR}YOU LOSE THE BATTLE"
420 PRINT"IF YOU:"
430 PRINT"{DOWN}  Q ARE BLASTED BY A"
440 PRINT"          MINGON'S LASER"
450 PRINT"{DOWN}  Q ARE SUCKED INTO A"
460 PRINT"          BLACK HOLE"
470 PRINT"{DOWN}  Q COLLIDE WITH AN"
480 PRINT"          ASTEROID"
490 PRINT"{DOWN}  Q ARE PULLED INTO"
500 PRINT"          THE SUN BY GRAVITY"
510 PRINT"  Q ARE RAMMED BY THE"
520 PRINT"          MINGON SHIP"
530 PRINT"{DOWN}PRESS ANY KEY"
540 A$="":GETA$:IFA$=""THEN540
2000 PRINT"{CLEAR}  ENTER SKILL LEVEL"
2010 PRINT"{02 DOWN}      {REV}1{OFF} - TRIV
      IAL"
2020 PRINT"{DOWN}          TO"
2030 PRINT"{DOWN}      {REV}9{OFF} - IMPOSSI
      BLE"
2040 A$="":GETA$:IFA$=""THEN2040
2050 SL=VAL(A$):IFSL<1ORSL>9THEN2000
2100 PRINT"{CLEAR}  SPACEWAR OPTIONS"
2110 PRINT"{02 DOWN}      {REV}S{OFF}UN'S GR
      AVITY"
2120 PRINT"{DOWN}      {REV}B{OFF}LACK HOLE"
2130 PRINT"{DOWN}      {REV}N{OFF}O GRAVITY"
2140 PRINT"{02 DOWN}  SELECT ONE OPTION":G
      =0
2150 A$="":GETA$:IFA$=""THEN2150
2160 IFA$="S"THENG=1
2170 IFA$="B"THENG=2
2180 POKES-2,SL:POKES-1,G

```

```
2185 PRINT"{02 DOWN}JUST A FEW MOMENTS"
2190 POKE198,5:POKE631,78:POKE632,69:POKE6
    33,87:POKE634,13:POKE635,131:END
```

```
9800 DATA7168,192,240,127,103,34,54,63,50
9810 DATA7176,4,14,62,227,227,62,14,4
9820 DATA7184,50,63,54,34,103,127,240,192
9830 DATA7192,24,24,60,36,102,231,126,24
9840 DATA7208,24,126,231,102,36,60,24,24
9850 DATA7216,3,15,254,230,68,108,252,76
9860 DATA7224,32,112,124,199,199,124,112,3
    2
```

```
9870 DATA7232,76,252,108,68,230,254,15,3
9871 DATA7240,0,0,12,28,56,48,0,0
9872 DATA7248,0,0,48,56,28,12,0,0
9873 DATA7256,0,24,24,24,24,24,0,0
9874 DATA7264,0,0,0,62,62,0,0,0
9900 DATA7296,0,112,126,102,32,48,48,0
9910 DATA7304,228,18,37,68,36,18,33,198
9920 DATA7312,0,48,48,32,102,126,112,0
9930 DATA7320,68,170,145,0,34,85,137,129
9940 DATA7328,153,90,60,255,255,60,90,153
9950 DATA7336,129,137,85,34,0,145,170,68
9960 DATA7344,0,14,126,102,4,12,12,0
9970 DATA7352,198,33,18,36,68,37,18,228
9980 DATA7360,0,12,12,4,102,126,14,0,-1
```

Program 3. Spacewar Part 2.

```
1 C=22:R=23:S=7680:A=30720:DD=37154:P1=37151
    :P2=37152:SL=PEEK(S-2)
2 POKE36879,27:T=3:CY=4:CE=6:CA=1:CS=7:CQ=1:
    CM=0:X=-1:G=PEEK(S-1)
3 V=36878:S1=V-2:S2=V-1:DIMDC%(2,2),OC%(2,2)
4 FORI=0TO2:FORJ=0TO2:X=X+1:DC%(I,J)=X:OC%(I
    ,J)=X+16:NEXTJ,I
5 DEFFNA(Z)=S+X+C*Y:DEFFNB(Z)=PEEK(FNA(Z)):D
    EFFNR(Z)=INT(RND(1)*Z)
6 SX=50:SY=50:POKEV-9,255:PRINT"{CLEAR}":X=S
    +A:FORI=XTOX+505:POKEI,T:NEXTI
7 POKE36879,62:FORI=1TO3+2*SL:X=FNR(C):Y=FNR
```

```

(R)
8 POKEFNA(0),46:POKEFNA(0)+A,CA:NEXT
9 IFG=1THENX=11:Y=12:POKEFNA(0),20:POKEFNA(0
)+A,CS: SX=X:SY=Y
10 IFG=2THENSX=FNR(C):SY=FNR(R)
11 X=FNR(C):Y=FNR(R):IFFNB(0)<>32THEN11
12 D=X:E=Y:Q=FNA(0):U=-1:O=FNR(3)-1
13 POKEFNA(0),DC%(U+1,O+1):POKEFNA(0)+A,CY
14 X=FNR(C):Y=FNR(R):IFFNB(0)<>32THEN14
15 H=X:L=Y:K=FNA(0):M=1:N=FNR(3)-1
16 POKEFNA(0),DC%(M+1,N+1):POKEFNA(0)+A,CE
17 A$="":GETA$:IFA$=" "THEND=FNR(C):E=FNR(R)
18 GOSUB75:IFFBTHENX=D:Y=E:PX=U:PY=O:GOTO47
19 B=0:F=0:IFJ0THENB=1

20 IFJ2THENB=-1
21 IFJ1THENF=1
22 IFJ3THENF=-1
23 IFB=0ANDF=0THEN B=U:F=0
24 U=B:O=F:IFG=0THEN28
25 B= SX-D:F=SY-E:J=SQR(B*B+F*F):J=(20-J)/30:J
=1-J*J
26 IFRND(1)<JTHEN28
27 B=SGN(B):F=SGN(F):D=D+B:E=E+F:GOTO29
28 D=D+U:E=E+O
29 IFE<0THENE=R
30 IFE>RTHENE=0
31 IFD>CTHEND=0
32 IFD<0THEND=C
33 X=D:Y=E:J=FNB(0):IFJ=32THEN35
34 IFJ=46ORJ=20ORFNA(0)=KTHEN60
35 IFX= SXANDY=SYTHENA$="WERE SUCKED INTO A BL
ACK HOLE!":WC=WC+1:GOTO71
36 POKEQ,32:POKEQ+A,T:Q=FNA(0):POKEQ,DC%(U+1,
O+1):POKEQ+A,CY
37 J=0:IFFNR(9)>SLTHEN39
38 M=D-H:N=E-L:M=SGN(M):N=SGN(N):IFM=0ORN=0TH
ENJ=1
39 H=H+M:L=L+N:IFH<0THENH=C
40 IFH>CTHENH=0
41 IFL>RTHENL=0
42 IFL<0THENL=R
43 X=H:Y=L:IFFNB(0)<>32THENM=FNR(3)-1:N=FNR(3
)-1:GOTO39

```

CHAPTER ONE

```
44 POKEK,32:POKEK+A,T:K=FNA(0):POKEK,DC%(M+1,
    N+1):POKEK+A,CE
45 IFRND(1)<0.1OR(J=1ANDFNR(9)<SL)THENPX=M:PY
    =N:GOTO47
46 GOTO17
47 Z=PX*PY:POKEV,8:IFZ=1THENJ=10
48 IFZ=-1THENJ=9
49 IFZ=0ANDPX=0THENJ=11
50 IFZ=0ANDPY=0THENJ=12
51 FORI=1TO10:X=X+PX:Y=Y+PY:POKES2,230-I
52 IFI<>1THENPOKEZ,32:POKEZ+A,T:IFX>CTHENX=0
53 IFX<0THENX=C
54 IFY>RTHENY=0
55 IFY<0THENY=R
56 B=FNB(0):IFB=32THEN58
57 IFB=46ORB=20ORFNA(0)=KORFNA(0)=QTHENI=10:N
    EXTI:GOTO60
58 Z=FNA(0):POKEZ,J:POKEZ+A,CM:NEXTI
59 POKEZ,32:POKEZ+A,T:POKEV,0:GOTO17
60 POKES2,230:SC=X-1:IFSC<0THENSFC=0
61 FC=X+1:IFFC>CTHENFC=C
62 SR=Y-1:IFSR<0THENSRR=0
63 FR=Y+1:IFFR>RTHENFR=R
64 FORX=SCTOFC:FORY=SRTOFR:J=OC%(X-SC,Y-SR)
65 POKEFNA(0),J:POKEFNA(0)+A,CQ:NEXTY,X
66 POKES1,220:FORJ=15TO0STEP-1:POKEV,J:FORJ1=
    1TO50:NEXTJ1:NEXTJ
67 POKEV,0:FORX=SCTOFC:FORY=SRTOFR:POKEFNA(0)
    ,32:POKEFNA(0)+A,T:NEXTY,X
68 IFPEEK(Q)=32THENA$="WERE VAPORIZED!":WC=WC
    +1:GOTO71
69 IFPEEK(K)=32THENA$="TRIUMPHED!":WH=WH+1:GO
    TO71
70 GOTO17
71 POKEV-9,240:PRINT"{CLEAR}{03 DOWN}YOU ";A$
72 PRINT"{02 DOWN}SCORE: VIC";WC;" YOU";WH
73 INPUT"{02 DOWN}PLAY AGAIN Y{03 LEFT}";A$:
    IFA$="Y"THEN6
74 END
75 POKEDD,127:P=PEEK(P2)AND128:J0=-(P=0):POKE
    DD,255
76 P=PEEK(P1):J1=-(PAND8=0):J2=-(PAND16=0)
77 J3=-(PAND4=0):FB=-(PAND32=0):RETURN
```

Extended Input Devices: Paddles And The Keyboard

MIKE BASSMAN / SALOMON LEDERMAN

You can use game paddles with the VIC, even from a BASIC program. Also included is information on VIC's "polled keyboard."

The VIC-20 has some remarkable capabilities not documented by the manual. Specifically, you can use game paddles with the VIC-20 as well as make better use of the keyboard.

What A Paddle Does

Have you ever seen the little nine-pin port right next to the power switch? This port can be used with paddles. To make life easy, it can be used with the widely available Atari game paddles (which are used with their video games and home computers). Just plug in a pair, and we'll be ready to begin. These paddles are *linear* devices. This means the paddle is a much more sensitive device than a directional joystick, which can only point in eight or so directions. You may think the paddle is not even as good, pointing only left or right. This is not true.

What the paddle actually does is isolate one position out of the 256 possible ones. When the paddle is turned to the far right, this value is 0. Every time you turn the paddle in either direction the number is increased or decreased accordingly. The VIC-20 allows up to two paddles. For each of them, we can obtain a position value. These values are in memory locations \$9008 for the first paddle, and \$9009 for the second. In decimal these are 36872 and 36873, respectively. (A number preceded by a "\$" signifies it is hexadecimal.)

How To Do It

Shown below is a quick one-liner that prints out the values of both the paddle registers.

```
10 PRINT PEEK(36872);PEEK(36873);GOTO 10
```

CHAPTER ONE

Try typing and running this program now. You should see a continuous stream of two numbers flying by. Fiddle with the paddles. The numbers should change accordingly. The more you turn a paddle left, the higher the number goes (the opposite for right, of course).

Next, we'll try something a little more complicated and which might be more applicable. Program 1 will move a little ball across the screen according to your paddle position. It will also slide a musical tone up and down at the same time.

The first two lines are just set-up, setting volume for the tone generator and clearing the screen. Line 20 gets the initial paddle position. The next line, 30, determines the ball's position on the screen. The ball can move from the far left edge of the screen (memory location 7900) to the far right (7921). Logically, the thing to do is to move the ball a little bit left whenever the paddle value goes a little bit up (turning towards the left). The problem is that the paddle is much more accurate than one line of the screen.

While the paddle has 256 possible calibrations, one line of the screen is only 22 characters long. What we do is to make a proportion of paddle calibrations per screen character, in this case 11.64 (obtained by dividing 256 by 22). Now we have the position of the ball on the screen. Line 40 does almost the same thing, finding an appropriate tone for the paddle position. We have 128 possible tones, so the proportion of the calibrations to tones is only 2 to 1. The next three lines just put the ball on the screen, tack a color onto it, and turn on the proper tone. Only the clean-up work remains to be done now: a small delay loop so the ball doesn't flicker badly, and erasing the ball and the color. After this, we get a new paddle reading and start all over again.

If you have run this little demonstration, the advantage of a linear device should be obvious. You can just whip the paddle back and forth without having to worry whether the computer is fast enough to keep up with you, and the ball will follow because the paddles determine an absolute position, rather than just a direction. This could be very convenient in games where speed is indispensable. In the near future, I'm sure we'll see many clever and innovative ways to use the paddles.

The Keyboard

There are two types of keyboards: ASCII, or hardware keyboards, and *polled*, or software keyboards. The ASCII keyboard is a separate device from the computer, which just sends out the ASCII value of

the key being pressed. The polled keyboard is a little more subtle. A polled keyboard is split up into sections of eight keys, called rows. Generally, a polled keyboard has eight rows. The computer can test one row at a time, and detect which key along which row is being pressed, if any. The polled keyboard can also detect any number of key combinations along any particular row. Consequently, polled keyboards need a fair amount of system software to do what comes naturally to an ASCII keyboard.

Most microcomputers today, the VIC-20 included, use polled keyboards because of the added flexibility and lower price. Unfortunately, the VIC-20 normally does not let us get at some of those nice features. To us, from BASIC, it seems just like an ASCII keyboard. We can only obtain one character at a time using the GET command. If two keys are being pressed down at once, the GET command will almost randomly choose one of those two as the value that gets sent back to the user. If you wanted to do a two-player game or a game requiring simultaneous depressing of more than one key, life would be very difficult. But here's how it can be done.

Polled Keyboard Encoding

The VIC-20 polled keyboard has eight rows of eight keys each. Each row can be selected by a particular value. The eight values for the eight rows are all shown in Table 1. These values are by no means arbitrary. If you examine the table, you can see that the values are given in binary, as well as decimal and hexadecimal. Row values were made by turning on all the bits in the byte, then turning off the bit which the row represents. For example, the first row has all the bits on (set to 1) except for the one on the far left, which is off (or 0). Then this binary number is simply used in its hexadecimal or decimal form to represent the row. Each key along the row is handled in exactly the same manner as the rows (for example, the value representing the first row would be the same as the one representing the first key in that row). This is a little confusing, but it works out well in the end. Table 2 is the keyboard encoding matrix. It shows all the row values going down, and all the keys along each row, and their values. For instance, the keys on row 223 are F3, =, ;, K, H, F, S, and Commodore. The value of the Commodore key would be 254.

Implementing Keyboard Theory

Using an individual row on the keyboard is accomplished as follows. You select a row by POKÉing its value into a memory location we'll call the row select register. Then you can get the information as to which key(s) is hit by PEEKing another location, the keyboard data

CHAPTER ONE

register. The row select register is located at \$9120 (37152), and the data register at \$9121 (37153).

Things don't work out as easily as doing just one POKE, then another PEEK. The problem, in this case, is the RUN STOP routine. This part of BASIC checks if you hit this key during execution of a program. If you have, the program stops. After every command executed, the routine puts a 247 in the row select register (the row which has the RUN STOP key) and checks the data register for a value of 254 (eighth key over). If the data register is 254, then you have hit the RUN STOP key, and program execution terminates.

This means that even after we have just chosen a row by POKEing a value into the select register, the RUN STOP routine will change it right back to a 247. Very bad news indeed, unless you only want to use row 247. Not only that, but you can't use the RUN STOP key for your own purposes. There is a way to disable the RUN STOP key. POKEing 808 with 114 turns off the RUN STOP key, and POKEing 808 with 112 turns it back on again. This does not solve our problem. Turning off the RUN STOP key will prevent it from ending program execution when that key is hit, but the routine still stores that 247 in the select register. However, when we clear up the major problem, turning off the RUN STOP key will allow us to use that key in our programs.

A Solution

The way to solve this problem is by noticing that this routine operates after every BASIC command. What must be done is to POKE in our select value, then PEEK the data register, all in the time of less than one BASIC command. Machine language is the answer. The VIC-20 can use machine language even though it has no direct facilities for entering or saving it. [See *Jim Butterfield's TINYMON1*, reprinted in this book, which provides a monitor for VIC.] We are going to use a very short machine language routine that simply puts our row into the select register, looks at the data register, then puts the contents of the data register into a RAM location into which the BASIC program can look. Program 2 shows just such a machine language program. Not much to it at all, just five lines of code. The first instruction loads in the row value, in this case a \$7F (127). The second stores it in the select register. The next picks up a value from the data register. The last two just store that value in accessible RAM (at \$1DFF, or 7679), then return to BASIC.

This routine will do the trick because it does what we want in less than one BASIC command. Even though the VIC-20 has no real

method for entering machine code data, it can be done anyway. You just take the machine code values, convert them into decimal, and stick them into a BASIC DATA statement. Then just add a line of BASIC that reads the values and puts them in the correct place. In Program 3, we have a complete demonstration. Lines 30 and 40 are the aforementioned DATA statement and reader/POKEr. Line 5 turns off the RUN STOP.

Lines 10 and 20 need a little bit of explanation. We are going to put the machine language routine into the top of available memory. Unfortunately, BASIC also wants to use this space. These lines tell BASIC not to use the highest 21 bytes of RAM. Locations 51 and 52, as well as 55 and 56, contain the top of BASIC RAM in low, high format. Low, high format is when the low byte of an address precedes its high byte. To calculate an address from this format, just use this formula: $(256 * \text{high byte} + \text{low byte} = \text{address})$. Normally the low and high bytes for the top of BASIC are 00 and 30, respectively (yielding an address of 7680). These we change to 235 and 29, giving an address of 7659. Line 50 goes to our machine code routine, line 60 prints the result, and 70 repeats the process. Try it now. I'll wait. If you press one of the keys from the first row, the appropriate value will be printed. No key is indicated by its printing 255. As it is now, this program will print first row values. To change the row, just change the second item of data in line 30. I used this program, incidentally, to make the keyboard matrix chart.

All this may seem pretty useless to you at this point. Our next program will do something that cannot be done with regular old BASIC. Program 4 will play a tone of varying pitch depending on which of two keys you hit. Doesn't sound too exciting, but it will play the two tones one after the other even if both keys are pressed at the same time. This is the basis of two-player games, where the computer can fairly give one turn to each player. All the material in this program should be old hat to you now, so I won't bother to explain it.

Possibly you've learned to use your paddles and keyboard now. Put them to good use!

CHAPTER ONE

Table 1.

127 - 7F - 0111 1111
191 - BF - 1011 1111
223 - DF - 1101 1111
239 - EF - 1110 1111
247 - F7 - 1111 0111
251 - FB - 1111 1011
253 - FD - 1111 1101
254 - FE - 1111 1110

Table 2. Keyboard Matrix Table

Column
(POKE)
↓

Row (PEEK)
of
Keys
→

	127	191	223	239	247	251	253	254
127	F7	Home	—	Ø	8	6	4	2
191	F5		@	O	U	T	E	Q
223	F3	=	:	K	H	F	S	COMMO- DORE
239	F1	RIGHT SHIFT	.	M	B	C	Z	SPACE
247	CURSOR	/	,	N	V	X	LEFT SHIFT	RUN STOP
251	CURSOR	;	L	J	G	D	A	CTRL
253	RETURN	*	P	I	Y	R	W	
254	DEL	£	+	9	7	5	3	1

Program 1.

```
5 POKE36878,3
10 PRINT"{CLEAR}"
20 X=PEEK(36872)
30 L=7921-INT(X/11.64)
40 T=255-INT(X/2):IFT=255THENT=254
50 POKEL,81
55 POKEL+30720,2
```

```
60 POKE36874,T
70 FORK=1TO10:NEXT
80 POKE1,32:POKE1+30720,1
90 GOTO20
```

Program 2.

```
A9 7F      LDA #$7F
8D 20 91    STA $9120
AD 21 91    LDA $9121
8D FF 1D    STA $1DFF
60          RTS
```

Program 3.

```
5 POKE808,114
10 POKE51,235:POKE52,29
20 POKE55,235:POKE56,29
30 DATA169,127,141,32,145,173,33,145,141,255,
   29,96
40 FORK=1TO12:READX:POKE7659+K,X:NEXTK
50 SYS7660
60 PRINTPEEK(7679);
70 GOTO50
```

Program 4.

```
10 POKE808,114:POKE51,235:POKE52,29:POKE55,23
   5:POKE56,29:POKE36878,3
20 DATA169,127,141,32,145,173,33,145,141,255,
   29,96
30 FORK=1TO12:READX:POKE7659+K,X:NEXTK
40 POKE7661,127:SYS7660
50 IFPEEK(7679)=254THENPOKE36874,200:FORK=1TO
   500:NEXTK:POKE36874,0
60 POKE7661,191:SYS7660
70 IFPEEK(7679)=127THENPOKE36875,200:FORK=1TO
   500:NEXTK:POKE36875,0
80 GOTO40
```

Game Paddles

DAVID MALMBERG

How to use paddles with the VIC – with two classic games, Pong and Breakout.

VIC can use the Atari game paddles. This article is a tutorial on how these paddles work with the VIC with a detailed discussion of a Pong game. This version of Pong can have two human players against one another, or one player against the VIC, which has nine skill levels. A game paddle version of the classic game Breakout with three skill levels is also presented. After studying these two programs, the reader should be a game paddle "expert" and be capable of easily incorporating game paddles into his own programs.

How Game Paddles Work

For those readers who are unfamiliar with the Atari game paddles, a brief description is in order. These game paddles are included when you buy an Atari home video computer system (the game machine), or may be purchased separately as a peripheral device for the Atari personal computer. The price for a pair of paddles varies between \$15 and \$20. There are two separate paddle units which attach to a single connector that plugs directly into the VIC game port. Each paddle unit consists of a red "fire" button and a knob that may be turned freely in either direction. The knob is attached to a potentiometer which varies a voltage fed into the VIC's game port. After converting this voltage to a digital value, the VIC is able to know the exact position to which the knob has been turned.

To see how the VIC can read the paddles, plug your paddles into the game port and enter the following short program:

```
10 DD=37154:P1=37151:P2=37152
20 PX=36872:PY=36873
30 POKE DD,127:P=PEEK(P2)AND128
40 FR=-(P=0):POKE DD,255
50 P=PEEK(P1):FL=-(PAND16)=0
60 VL=PEEK(PX):VR=PEEK(PY)
70 PRINT"{CLEAR}";FL;VL;FR;VR
200 GOTO 30
```

When you RUN this program you should see four numbers in the first row of the screen. The first two numbers correspond to the left paddle and the last two values to the right paddle. FL and FR will be either one or zero depending upon whether the left or right "fire" button is pushed or not. VL and VR will correspond to the knob settings for the left and right paddles respectively. Both VL and VR will vary between 255 and zero as the knobs are turned clockwise from their leftmost position to their rightmost position. You will notice that the knobs are more sensitive than their wide arc would imply; i.e., there is a large band of arc at either extreme of the knob's movement where the values stay at 255 or zero. The actual arc where the values change linearly from zero to 255 is only about one-quarter of a full turn. Although this is a little more sensitive than you might wish, you will find that this will still be enough arc to produce some exciting games.

The reason why this short program actually works is beyond both the scope of this article and the interest level of most readers. I will leave it to Commodore to explain more fully when they issue their documentation on the game paddles. Suffice it to say, it does work!

Controlling Screen Motion

To see how the game paddles can be used to control motion on the VIC's screen, *add* the following lines of code to the above program:

```

2 POKE 36879,27:PRINT"{CLEAR}":C=4
4 S=256*PEEK(648):A=30720:LL=S
6 IFPEEK(648)=16THENA=33792
70 X=21-INT(VL*21/255):Y=22-INT(VR*22/255)
80 NL=S+X+22*Y:CL=NL+A
90 IF FR OR FL THEN C=C+1
100 IF C=8 THEN C=2
110 IF NL=LL THEN 30
120 POKE LL,32:POKE LL+A,1
130 POKE NL,81:POKE CL,C
140 LL=NL

```

You will notice that line 70 above replaces line 70 in the previous program. All of the other lines are new additions.

When you run this program, you will find that the game paddles will move a ball graphic character around the screen at a very rapid pace. Specifically, the left paddle will move the ball from left to right horizontally, and the right paddle will cause vertical movement from

top to bottom. Pushing either one of the fire buttons will change the ball's color.

Let's look at this short program in more detail. It not only demonstrates how the game paddles can control motion (and will make following the logic of Pong and Breakout easier), but also contains several useful techniques that will help improve any "action" game you may write. Line 2 sets the border color to cyan and the background color to white, and clears the screen. The variable "C" contains the color of the ball and is initialized to purple.

Lines 4 and 6 will be a useful addition to any program that POKEs characters to the screen. As you may have read or even experienced, when memory modules (8K or more) are plugged into the VIC, the locations of the screen memory and the color matrix move. When you run a program which assumes the screen is in its "normal" position of 7680 and the screen has actually moved (to 4096), the results are frequently a disaster. Lines 4 and 6 determine the correct locations for the screen and the color matrix – regardless of the amount of memory that has been added to the VIC. In line 4, the variable "S" will be the starting location of the screen and will be either 7680 or 4096. The variable "A" is the value that must be added to a particular screen location to get its corresponding color matrix location. For example, if we POKE location S with 81 (a ball character), and we POKE location S + A with 4, we would get a purple ball in the "home" corner of the screen. For a "normal" 3.5K VIC, "A" is 30720; i.e., the start of the color matrix is 7680 + 30720 or 38400. If the screen moves to 4096, the color matrix will move to 37888 and the value of "A" changes to 33792 – this is the condition given in line 6.

The logic of the Pong program actually starts on line 280, which defines a group of variables that will be used repeatedly later. R is the number of rows. C is the number of columns. NAS(1 and 2) contain the two players' names. SPS(1 and 2) contain strings with the cursor control characters needed to position the cursor where each player's name is printed. SC(1 and 2) contain the scores for the two players. In lines 290 and 300 more useful variables are defined. E and F are values used in the calculation of the paddle location. By making these calculations once at the start of the program, and just referencing their variable names later, the speed of the game is increased. The other variables in these lines are either identical to those used in the previous example or their purpose will be obvious when you see how they are used. Line 300 determines the starting locations for the screen, S, and color matrix, S + A.

Line 320 defines three very useful functions. FNA will return the current screen location corresponding to row Y and column X. This will normally be the ball's location. FNB will return the color matrix location corresponding to FNA. FNR(Z) will return a random integer between 0 and Z. Lines 330 to 470 ask the player(s) to specify the options for the play of the game. The variable N is the number of players. If $N = 1$ then the VIC will play the part of the player on the right side of the screen. The variable D is the skill level for the VIC. A value of one will play a very poor game. A value of nine will never miss a shot.

Lines 480 to 520 begin the game by zeroing both scores, randomly deciding who serves first, clearing the screen, drawing the border in row 1, and printing the names and scores in row 0. Line 530 tests the variable SV (which will either be 1 for the left player's serve or 2 for the right player's serve) and branches accordingly.

Lines 540 to 580 handle the left player's serve. Specifically, line 550 reads the status of the left fire button and allows the left paddle and the right paddles to move by the GOSUB's to 120 and 190 respectively. Line 560 jumps back to 550 unless the fire button has been pushed and the serve has been made. Line 570 randomly causes the serve to go upward ($DY = -1$) or downward ($DY = +1$). Line 580 puts the ball in front of the current position of the left paddle, sets its forward direction toward the right ($DX = +1$), sounds a "hit," and branches to line 670, which is the actual play loop.

Lines 590 to 650 are the right player's serve routines. Line 600 tests N for the number of players. If $N = 1$ and the VIC is playing the right side, line 610 waits a random amount of time and then serves. Lines 620 to 650 handle a human player on the right side almost identically to the way that lines 550 to 580 handle the left side player.

Line 670 is the start of the play loop and moves the ball one position in its current direction, via the GOSUB70. Line 690 allows the left paddle to move. Line 700 tests whether it is possible that the ball is about to hit a paddle because it is in either column 2 or column 19, i.e., one column from the paddles. If the answer is no, the ball is allowed to move one more position. This "extra" move in the playing loop makes the game considerably faster – to see how much faster, try the game with line 700 deleted.

Lines 710 to 790 handle the right player's paddle movement. Lines 730 to 760 are for the VIC playing the right paddle. Based on whether a random integer from zero to nine is greater than the skill level, the VIC will move. If the VIC moves, it moves so its paddle is

CHAPTER ONE

even vertically with the current position of the ball. Obviously, if the skill level is nine, then the VIC will always move and will never miss the ball. Line 780 makes the paddle move for a human player on the right side.

Line 790 again tests whether the ball is possibly going to hit a paddle because it is in a column next to a paddle. If the answer is no, then the program branches back to the start of the play loop at line 670. If the answer is yes, the ball is just about either to hit the paddle or to miss it. Lines 810 and 820 determine the values for ZZ, the index for the player who will win the point if there is a mix, and ZC, the screen PEEK/POKE character for the paddle. Line 830 slows any "fast" ball down to normal speed.

Lines 840 and 850 test if the ball would hit the paddle if it moved one more position. The variable Q in line 840 is the screen location of the position next to (in the same row) the ball's current position. If the screen character at location Q is equal to ZC, the appropriate paddle character, the ball is about to hit the middle of the paddle, and the program branches to 960. Line 850 performs the same type of test, but for a possible "corner" hit.

If the ball failed both of these tests, it is just about to miss the paddle. In this case lines 870 to 940 move the ball off the field, sound a "miss," update the score, determine the next server (the loser), and branch back to serve again or ask about a new game if that was the winning point.

If the ball is hit by the paddle, line 960 will change its direction, sound a "hit," and move it one position on its flight path. Line 970 will randomly make it a "fast" ball by doubling the ball's X movement, DX. This not only doubles the speed, but also changes the angle of flight. These "fast" balls make the game much more exciting.

After you have gone through the previous example, the subroutines used in Pong should seem very straightforward. The subroutine at line 70 checks if the ball is about to hit a wall and, if the answer is yes, it causes the ball to bounce. Then this subroutine erases the current ball location and draws it at its new location. The subroutines at 120 and 190 move the left and right paddles respectively. They first check to see if the paddle has moved since the last reading. If the answer is no, they RETURN. If yes, these subroutines erase the last paddle, and draw a paddle at the new location. The subroutine at 260 sounds a "hit" and the subroutine at 270 a "miss."

Breakout

The version of Breakout here also uses the game paddles. You will

find it much faster and more exciting than versions which use a joystick or the keyboard and can move the paddle only a column at a time. It has three skill levels and the highest will challenge even the most seasoned arcade malingerers.

The overall program flow and even the variable names are almost identical to Pong. The program is well commented and self-documenting.

Breakout

```

10 REM VIC BREAKOUT USING GAME PADDLE
20 REM BY DAVID MALMBERG
30 REM 43064 VIA MORAGA
40 REM FREMONT, CALIFORNIA 94538
50 GOTO200
60 REM SUB TO MOVE BALL ONE POSITION
70 IFX=0ORX=21THENDX=-DX:GOSUB190:REM BOUNCE ~
  IF WALL
80 POKEFNA(0),32:POKEFNC(0),1:REM ERASE CURRE
  NT BALL LOCATION
90 X=X+DX:Y=Y+DY:POKEFNA(0),81:POKEFNC(0),4:R
  EM DRAW NEXT BALL LOCATION
100 RETURN
110 REM SUB TO UPDATE PADDLE
120 VL=PEEK(PX):REM READ PADDLE
130 NL=E-INT(VL*F):IFNL=LLTHENRETURN:REM SAME ~
  AS LAST LOCATION
140 Z=S+461+LL:FORI=1TOD:POKEZ+I,32:POKEZ+I+A,
  1:NEXT:REM ERASE OLD PADDLE
150 Z=S+461+NL:FORI=1TOD:POKEZ+I,226:POKEZ+I+A
  ,6:NEXT:REM DRAW NEW PADDLE
160 LL=NL:REM UPDATE PADDLE LOCATION
170 RETURN
180 REM SUB TO SOUND HIT
190 POKEV,15:POKES1,220-3*Y:FORI=1TO50:NEXT:PO
  KEV,0:POKES1,0:RETURN
200 POKE36879,27:C=22:HI=0:X=RND(-TI)
210 REM INPUT PADDLE WIDTH (DIFFICULTY)
220 PRINT"{CLEAR}  WELCOME TO BREAKOUT"
230 PRINT"{DOWN}DO YOU WANT A"
240 PRINT"{DOWN}    {REV}1{OFF} - DIFFICULT"
250 PRINT"      {REV}2{OFF} - AVERAGE, OR"
260 PRINT"      {REV}3{OFF} - EASY GAME ?"

```

CHAPTER ONE

```
270 A$="":GETA$:IFA$=""THEN270
280 D=VAL(A$):IFD<10RD>3THEN270
290 E=C-D:F=E/255
300 V=36878:S1=36876:PX=36872:LL=8
310 S=256*PEEK(648):A=30720:IFPEEK(648)=16THEN
    A=33792
320 DEF FNA(Z)=S+X+C*Y : DEF FNC(Z)=FNA(Z)+A :
    DEF FNR(Z)=INT(Z*RND(1))
330 SC=0:BA=9
340 REM DRAW BRICKS
350 NN=0:PRINT"{CLEAR}":Y=1:FORX=0TO21:POKEFNA
    (0),160:POKEFNC(0),3:NEXT
360 FORY=5TO8:FORX=0TO21:POKEFNA(0),160:POKEFN
    C(0),Y-1:NEXTX,Y
370 PRINTCHR$(144)"{HOME}BA";BA;" HI";HI;" SC"
    ;SC
380 TT=FNR(300):ZZ=TI
390 IFTI-ZZ<TTTHENGOSUB120:GOTO390:REM RANDOM ~
    START
400 X=5+FNR(10):Y=9:DY=1:DX=-1:IFFNR(9)>4THEND
    X=1:REM NEW BALL DIRECTION
410 POKEFNA(0),81:POKEFNC(0),4:REM DRAW NEW BA
    LL
420 REM START OF MAIN LOOP
430 GOSUB70:REM MOVE BALL
440 GOSUB120:REM ERASE OLD PADDLE AND DRAW NEW
    ONE
450 REM TEST FOR PADDLE HIT OR MISS
460 IFY<>20THEN610:REM NOT NEAR PADDLE
470 Z=FNA(0)+C:IFPEEK(Z)=226THEN590:REM STRAIG
    HT HIT
480 IFPEEK(Z+DX)=226THENDX=-DX:GOTO590:REM COR
    NER HIT
490 REM MISS PADDLE
500 GOSUB70:GOSUB70:POKEFNA(0),32:POKEFNC(0),1
510 POKEV,15:FORI=1TO30:POKES1,200-2*I:NEXT:PO
    KEV,0:POKES1,0
520 BA=BA-1:PRINT"{HOME}BA";BA;" HI";HI;" SC";
    SC
530 IFBA<>0THEN380:REM NEXT BALL
540 PRINT"{HOME}{10 DOWN}      PLAY AGAIN ?"
550 A$="":GETA$:IFA$=""THEN550
560 IFA$<>"Y"THENEND
570 IFSC>HITHENHI=SC
```

```
580 GOTO330
590 GOSUB190:DY=-DY:GOSUB70:GOTO430:REM SOUND ~
    HIT AND MOVE
600 REM TEST FOR BRICK AREA
610 IFY<4ORY>9THEN710
620 REM IN BRICK AREA
630 IFPEEK(FNA(0)+C*DY+DX)<>160THEN430:REM NO ~
    BRICK NEXT - SO NORMAL MOVE
640 REM HIT BRICK NEXT
650 GOSUB70:GOSUB190:SC=SC+9-Y
660 NN=NN+1:IFNN>70THEN350:REM DRAW NEW BRICKS
670 PRINT"{HOME}BA";BA;" HI";HI;" SC";SC
680 IFFNR(Y+10)>4+3*DYTHENDY=-DY:GOTO430:REM B
    OUNCE BACK
690 GOTO610
700 REM TEST FOR TOP
710 IFY=2THENDY=-DY:GOSUB190:REM BOUNCE OFF TO
    P
720 GOTO430:REM END OF MAIN LOOP
```

Pong

```
10 REM VIC PONG USING GAME PADDLES
20 REM BY DAVID MALMBERG
30 REM 43064 VIA MORAGA
40 REM FREMONT, CALIFORNIA 94538
50 GOTO280
60 REM SUB TO MOVE BALL ONE POSITION
70 IFY=2ORY=22THENDY=-DY:GOSUB240:REM BOUNCE ~
    IF WALL
80 POKEFNA(0),32:POKEFNC(0),1:REM ERASE CURRE
    NT BALL LOCATION
90 X=X+DX:Y=Y+DY:POKEFNA(0),81:POKEFNC(0),4:R
    EM DRAW NEXT BALL LOCATION
100 RETURN
110 REM SUB TO UPDATE LEFT PADDLE
120 VL=PEEK(PX):REM READ PADDLE
130 NL=E-INT(VL*F):IFNL=LLTHENRETURN:REM SAME ~
    AS LAST LOCATION
140 Z=S+45+LL*C:FORI=0TO2:POKEZ+I*C,32:POKEZ+I
    *C+A,1:NEXT:REM ERASE OLD
150 Z=S+45+NL*C:FORI=0TO2:POKEZ+I*C,225:POKEZ+
    I*C+A,6:NEXT:REM DRAW NEW
```

CHAPTER ONE

```
160 LL=NL:REM UPDATE PADDLE LOCATION
170 RETURN
180 REM SUB TO UPDATE RIGHT PADDLE
190 VR=PEEK(PY):REM READ PADDLE
200 NR=INT(VR*F):IFNR=LRTHENRETURN:REM SAME AS
    LAST LOCATION
210 Z=S+64+LR*C:FORI=0TO2:POKEZ+I*C,32:POKEZ+I
    *C+A,1:NEXT:REM ERASE OLD
220 Z=S+64+NR*C:FORI=0TO2:POKEZ+I*C,97:POKEZ+I
    *C+A,6:NEXT:REM DRAW NEW
230 LR=NR:REM UPDATE PADDLE LOCATION
240 RETURN
250 REM SUB TO SOUND HIT
260 POKEV,15:POKES1,220-3*Y:FORI=1TO50:NEXTI:P
    OKEV,0:POKES1,0:RETURN
270 POKEV,15:FORI=1TO30:POKES1,200-2*I:NEXTI:P
    OKEV,0:POKES1,0:RETURN
280 POKE36879,27:R=23:C=22:X=RND(-TI):DIMNAS(2
    ),SP$(2),SC(2)
290 E=R-5:F=E/255:DD=37154:P1=37151:P2=37152:S
    P$(1)="{HOME}":SP$(2)="{HOME}{12 RIGH
    RIGHT}"
300 V=36878:S1=36876:PX=36872:PY=36873:LL=8:LR
    =8
310 S=256*PEEK(648):A=30720:IFPEEK(648)=16THEN
    A=33792
320 DEF FNA(Z)=S+X+C*Y : DEF FNC(Z)=FNA(Z)+A :
    DEF FNR(Z)=INT(Z*RND(1))
330 PRINT"{CLEAR} WELCOME TO VIC PONG"
340 PRINT"{DOWN} DO YOU WISH TO PLAY"
350 PRINT"{DOWN} {REV}1{OFF} - THE VIC, OR"
360 PRINT"{DOWN} {REV}2{OFF} - ANOTHER PLAYER ~
    ?"
370 A$="":GETA$:IFA$=""THEN370
380 N=VAL(A$):IFN<1ORN>2THEN370
390 PRINT"{DOWN}ENTER NAME(S) "
400 FORI=1TON:PRINT"{DOWN}PLAYER";I;:INPUTNAS(
    I):NEXTI
410 IFN<>1THEN490
420 NAS(2)="VIC":PRINT"{CLEAR}HOW HARD SHOULD ~
    I PLAY"
430 PRINT"{DOWN}          {REV}1{OFF} - EASY"
440 PRINT"{DOWN}          TO"
450 PRINT"{DOWN}          {REV}9{OFF} - IMPOSSIBLE"
460 A$="":GETA$:IFA$=""THEN460
```

```
470 H=VAL(A$):IFH<1ORH>9THEN460
480 REM BEGINNING SERVE AND GAME
490 B$="":NL=9:NR=9:SC(1)=0:SC(2)=0:
SV=1:IFFNR(9)>4THENSV=2
500 PRINT"{CLEAR}":Y=1:FORX=0TO21:POKEFNA(0),1
60:POKEFNC(0),3:NEXTX
510 FORJ=1TO3:PRINTSP$(SV)"SERVICE":GOSUB270:P
RINTSP$(SV)B$:GOSUB260:NEXTJ
520 PRINT"{HOME}";NA$(1);SC(1);SP$(2);NA$(2);S
C(2)
530 ON SV GOTO550,600
540 REM LEFT SERVE
550 POKEDD,255:P=PEEK(P1):JL=-((PAND16)=0):GOS
UB120:GOSUB190
560 IFJL<>1THEN550
570 DY=1:IFFNR(9)>4THENDY=-1
580 Y=NL+3:X=2:DX=1:GOSUB260:GOTO670
590 REM RIGHT SERVE
600 ON N GOTO610,620
610 NR=FNR(19):GOSUB210:GOTO640:REM VIC'S SERV
E
620 POKEDD,127:P=PEEK(P2)AND128:JR=-((P=0):GOSU
B190:GOSUB120:REM HUMAN SERVE
630 IFJR<>1THEN620
640 DY=1:IFFNR(9)>4THENDY=-1
650 POKEDD,255:Y=NR+3:X=19:DX=-1:GOSUB260
660 REM START OF PLAY LOOP
670 GOSUB70:REM MOVE BALL
680 REM LEFT MOVE
690 GOSUB120
700 IFX<>2ANDX<>19THENGOSUB70
710 REM RIGHT MOVE
720 ON N GOTO730,780
730 IFH<FNR(9)THEN790:REM COMPUTER MOVE
740 NR=Y-3:IFNR<0THENNR=0
750 IFNR>18THENNR=18
760 GOSUB210:GOTO790
770 REM HUMAN RIGHT MOVE
780 GOSUB190
790 IFX<>2ANDX<>19THEN670
800 REM TEST FOR PADDLE HIT OR MISS
810 IFX=2THENZZ=2:ZC=225
820 IFX=19THENZZ=1:ZC=97
830 IFABS(DX)=2THENDX=DX/2
840 Q=FNA(0)+DX:IFPEEK(Q)=ZCTHEN960:REM STRAIG
```


CHAPTER ONE

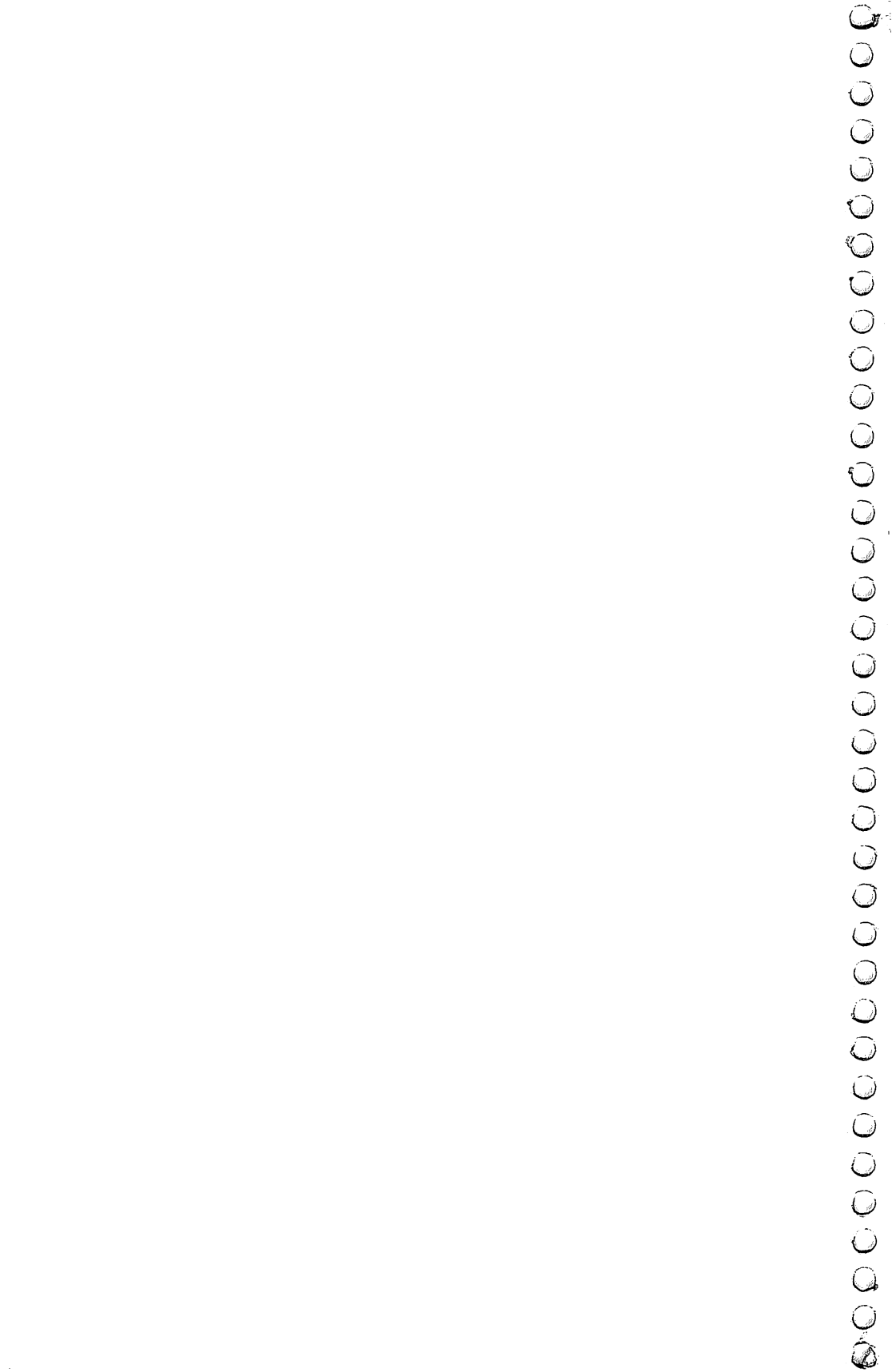
```
      HT HIT
850 IFPEEK(Q+C*DY)=ZCTHENDY=-DY:GOTO960:REM CO
    RNER HIT
860 REM MISS PADDLE
870 GOSUB70:GOSUB70:POKEFNA(0),32:POKEFNC(0),1

880 SC(ZZ)=SC(ZZ)+1:SV=2:IFZZ=2THENSV=1:REM UP
    DATE SCORE, LOSER SERVES
890 IFSC(ZZ)<>15THEN510:REM NEXT SERVE
900 FORJ=1TO10:PRINTSP$(ZZ)"WINNER!":GOSUB270:
    PRINTSP$(ZZ)B$:GOSUB260:NEXTJ
910 PRINT"{HOME}{11 DOWN}      PLAY AGAIN ?"
920 A$="":GETA$:IFA$=""THEN920
930 IFA$<>"Y"THENEND
940 GOTO490:REM NEW GAME
950 REM HIT PADDLE
960 DX=-DX:GOSUB260:GOSUB70:REM SOUND HIT AND ~
    MOVE
970 IFFNR(9)>4THENDX=2*DX:REM DOUBLE X-SPEED
980 GOTO670
```

CHAPTER TWO

DIVERSIONS— RECREATION AND EDUCATION





The Joystick Connection: Meteor Maze

PAUL L. BUPP / STEPHEN P. DROP

An exciting game serves to illustrate how joysticks are used with the VIC.

Let the games begin! Your VIC can be easily connected to the readily available Atari joysticks. We will show the new VIC-20 owner how to use these joysticks. Also there's a new VIC game called "Meteor Maze," which demonstrates the use of the joystick.

But first, let's look at how the joystick connects to the VIC. Program 1 is a BASIC joystick demonstration program. A line-by-line description of the program follows:

```
10 PRINT CHR$(147)
15 PRINT SPC(3) CHR$(95) "JOYSTICK
   DIRECTION"
20 PRINT SPC(3) CHR$(95) "BUTTON"
```

Lines 10 through 20 clear and print the display screen used by the program. CHR\$(95) prints the left arrow.

```
25 PRINT SPC(177) CHR$(144) "*** JO
   YSTICK DEMO ***"
```

Line 25 uses the SPC command to print 177 spaces, then a CHR\$(144) turns on the black print mode before printing the title of the program.

```
30 POKE 37154,127
```

This line resets the direction of the 6522 A side Data Direction register which was already set by the system to check the keyboard. The other Data Direction register used by the joystick is already set by the system default. (Note: With this register altered, some keys will now no longer be recognized by the system. See important note to line 90 below.)

CHAPTER TWO

```
35 PRINT CHR$(19)
```

This line homes the cursor to the top left of the screen.

```
40 A = ( PEEK(37137) AND 28) OR (P  
    EEK(37152) AND 128)
```

This line pulls together the two input register values used by the joystick and combines them to make a single value (A).

```
45 A = ABS((A-100)/4)-7
```

This line reduces the joystick value (Variable A) from line 40 to a simple number value between one and 13 with some number values (four and eight through ten) not being used. This value is kept in variable A. For each direction of the joystick, Figure 1 provides a visual display of the original value (boxed) and the condensed value placed in variable A by lines 40 and 45.

```
50 ON A GOSUB 100,110,120,,130,140  
    ,150,,,,160,170,180
```

This line directs the program to go to the chosen joystick direction subroutine and then to return to the next line of BASIC.

```
55 B = PEEK(37137)AND 32
```

Here the variable B is set to zero if the joystick button is pushed, or set to 32 if it is not being pushed. This PEEK is looking at only the one bit which shows whether the button has been pushed by the player.

```
60 PRINT CHR$(19)  
65 PRINT
```

These two lines home the cursor and move it down one line to place it at the right location on the screen to print whether the button is ON or OFF.

```
75 IF B GOTO 85
```

This IF statement only goes to line 85 if the variable B is not zero. In this case, it means the button was not pushed. Otherwise, the IF fails and the BASIC program proceeds to the next statement.

```
80 PRINT " ON": GOTO 90  
85 PRINT "OFF"
```

These lines print whether the button is ON or OFF depending on the IF statement in line 75.

90 POKE 37154,255

This line resets the Data Direction register altered in line 30. This internal system register is used to check the keyboard. With this POKE, all the keys are again recognized by the VIC-20. If, in your program, the STOP button is pushed, or for some other reason the program is accidentally stopped, this Data Direction register becomes correctly reset by using the RUN/STOP and RESTORE button combination or by using POKE 37154,255.

95 GET A\$: IF A\$ = "" GOTO 30

This line of BASIC provides a way to end the program by looking at the keyboard input buffer with the GET, and then, if no key has been pushed, the program branches back to line 30 to begin again.

Lines 100 through 180 are the direction indicator subroutines which print the direction indicator at the top of the screen. These are reached from line 50 above.

```
99 END
100 PRINT " SW": RETURN
110 PRINT " NW": RETURN
120 PRINT "  W": RETURN
130 PRINT "  S": RETURN
140 PRINT "  N": RETURN
150 PRINT "   ": RETURN
160 PRINT "  E": RETURN
170 PRINT " NE": RETURN
180 PRINT " SE": RETURN
```

This BASIC program is written in "portable" code which can be simply typed into the VIC-20. However, for those planning to include the Joystick Connection in their own programs, the routine can be speeded up and condensed to take up less room. A condensed version of the same routine is included in the Meteor Maze game (Program 2). That completes the discussion of the Joystick Connection and how it works.

Now, let's look at the game program "Meteor Maze."

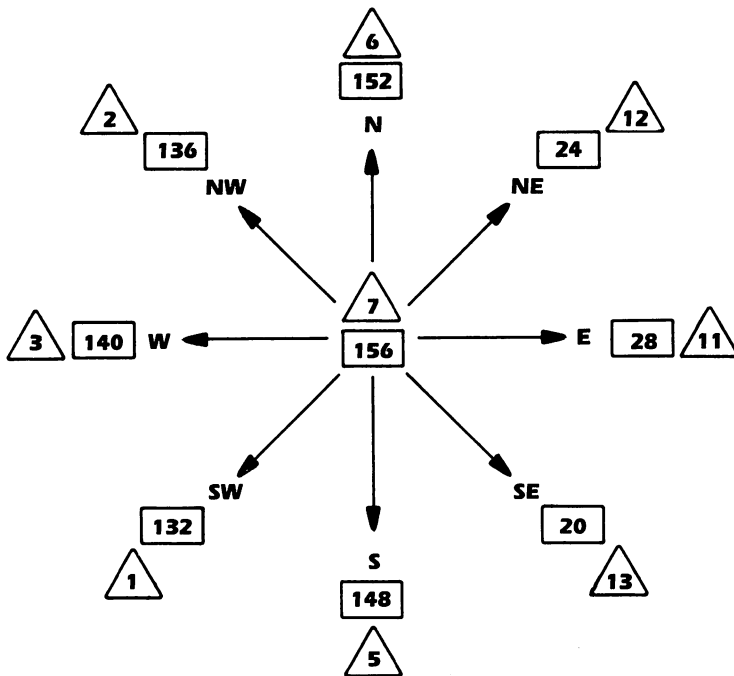
Meteor Maze

Meteor Maze is a fast game using the Joystick Connection routine

CHAPTER TWO

described above. The object of the game is to move your Scout Ship through the meteor field to the Base Ship at the bottom of the screen as quickly as possible. Two levels of play are available, Novice and Advanced. The difference in levels is the computer's tolerance for navigational error. Details of these differences can be displayed by pushing Function Key #1 (F1). Speed is of the essence. The player must learn to manipulate the controls well to be the fastest to reach the Base Ship.

Figure 1. Joystick Direction Values.



◻ ← VALUE returned to variable A in line 40 of Program 1.

△ ← VALUE returned to variable A in line 45 of Program 1.

Program 2.

```

2 PRINT "METEOR MAZE" SPC(96) "FOR I
  NSTRUCTIONS" SPC(34) "PRESS"
  SPC(40) "F1
3 POKE56,28:POKE52,28:CLR:FORA=71
  68TO7375:READB:POKEA,B:NEX
  T
4 V=36878:N=V-1:S=N-2:POKE37154,1
  27:L=7680:GOSUB97:F=13:IFG
  THENE=3:GOTO6
5 E=99
6 POKE36869,255:POKEV+1,27:PRINT"
  {OFF}{YEL}{CLEAR}V{BLK}";:
  FORA=1TO480:GETA$:IFA$=CHR
  $(133)GOTO40
7 PRINTMID$("@A@B@C@D@E@F",RND(TI
  )*12+1,1);:NEXT
8 PRINT"{RED}@LM{PUR}{REV} ELAPSE
  D TIME {OFF}{RED}NO{0
  7 LEFT}{REV}{PUR}";:POKE81
  85,16:POKE38905,2:TI$="000
  000"
9 A=(PEEK(37137)AND28)OR(PEEK(371
  52)AND128):A=ABS(A-100)/4-
  7:IFA=7THENA=F
10 ONAGOSUB90,91,92,,93,94,,,,,95,
  96,97:F=A
11 A=PEEK(37137)AND32:IFATHENPOKEV
  ,0:GOTO27
12 POKEN,255:POKES,220:POKEV,3:IFP
  EEK(C)>6GOTO27
13 IFPEEK(C)GOTO19
14 POKE30720+C,7:POKEC,PEEK(L):POK
  EL,0:L=C:IFC<>8161GOTO27
15 GOSUB99:POKE8161,22:PRINT"{13 L
  LEFT}DOCKING";:POKEV,15
16 FORL=1TO4:FORC=180TO235STEP2:PO
  KES,C:FORA=1TO10:NEXT:NEXT
  :POKES,0
17 FORA=1TO10:NEXT:READA,B:POKE816

```


CHAPTER TWO

```
1,A:POKE8162,B:NEXT
18 RESTORE:FORA=1TO208:READB:NEXT:
   K=0:GOTO4
19 ONGGOTO21
20 GOSUB99:POKEC,7:POKEN,255:FORA=
   15TO0STEP-1:POKEV,A:FORB=1
   TO35:NEXT:NEXT:GOTO27
21 GOSUB99:POKEN,220:FORA=16TO1STE
   P-1:POKEV,A:FORB=A*16-1TO(
   A-1)*16STEP-1:POKEV+1,B
23 NEXT:NEXT:POKE36865,132:POKEV+1
   ,59:POKE36869,242:K=K+1:GO
   SUB99
24 PRINT"{CLEAR}{BLK}{OFF}SCOUT"K"
   TO BASE:{DOWN}":PRINT"REQU
   EST"INT(EXP(K))"BOTTLES":P
   RINT"OF SUPER GLUE!
25 FORA=131TO0STEP-1:POKE36865,A:F
   ORB=1TO45:NEXT:NEXT
26 PRINT"{CLEAR}":GOSUB99:POKE3686
   5,25:GOTO4
27 PRINTMID$(TI$,3,2)": "MID$(TI$,5
   )"{05 LEFT}";
28 GETA$:IFA$=CHR$(133)GOTO40
29 IFA$<>CHR$(135)GOTO9
35 IFE=0ORPEEK(C)>8GOTO9
36 GOSUB99:POKEN,220:FORA=15TO0STE
   P-1:POKEC,8:POKEV,A:FORB=1
   TO20:NEXT:POKEC,0
37 FORB=1TO20:NEXT:NEXT:GOSUB99:E=
   E-1:GOTO9
40 POKEV+1,127:POKE36869,242:PRINT
   "{CLEAR}{OFF}{RED}      $$$
   $$$$$$"SPC(12)"{REV}SELECT
   ONE{BLK}{DOWN}
41 A$="      -ABLE TO DESTROY":PRINT"
   F1=NOVICE LEVEL":PRINTA$
42 B$="      -SHIP WILL ":PRINT"      9
   9 METEORS":PRINTB$"SURVIVE
   "SPC(5)"METEOR COLLISIONS{
   DOWN}
43 PRINT" F3=ADVANCE LEVEL":PRINTA
```

```
$SPC(7)"3 METEORS
44 PRINTB$"EXPLODE"SPC(5)"ON IMPAC
   T WITH A"SPC(6)"METEOR{DOW
   DOWN}
45 PRINT" F5=INSTRUCTIONS{DOWN}":P
   RINT" F7=END THE GAME{Ø2 D
   DOWN}":PRINT"* CURRENT LEV
   EL
46 IFGTHENPRINT"{HOME}"SPC(198)"*
   :GOTO48
47 PRINT"{HOME}{Ø3 DOWN}*"
48 POKE37154,255:GETA$:IFA$=""GOTO
   48
49 A=ASC(A$)-132:ONABS(A)GOTO51,52
   ,55,8Ø
5Ø GOTO4
51 G=Ø:GOTO4
52 G=1:GOTO4
55 POKEV+1,127:PRINT"{BLK}{CLEAR}{
   DOWN}GOAL-MOVE IN FRONT OF
   THE DOCKING BAY AND T
   HE BASE WILL LAND
56 A$=" YOUR SHIP.{DOWN}":PRINTA$
   :PRINT"JOYSTICK WILL POINT
   "A$
57 PRINT"FIRE BUTTON WILL MOVE "A$
58 PRINT"F1 ALLOWS CHANGE OF G
   AME DIFFICULTY.{DOWN}
59 PRINT"F5 DESTROYS METEORS T
   HAT ARE IN FRONT OF"A$

6Ø PRINT"{Ø2 DOWN}HIT A KEY TO CON
   TINUE
61 GETA$:IFA$=""GOTO61
62 GOTO4Ø
8Ø SYS4Ø96
9Ø POKEL,24:C=L+21:RETURN
91 POKEL,23:C=L-23:RETURN
92 POKEL,2Ø:C=L-1:RETURN
93 POKEL,18:C=L+22:RETURN
94 POKEL,17:C=L-22:RETURN
95 POKEL,19:C=L+1:RETURN
```

CHAPTER TWO

```
96 POKEL,21:C=L-21:RETURN
97 POKEL,22:C=L+23:RETURN
99 POKEV,0:POKEN,0:POKES,0:RETURN
1000 DATA0,0,0,0,0,0,0,0,0,12,62,127
    ,62,28,0,0,32,120,28,62,60
    ,24,0,0
1002 DATA24,60,126,124,60,56,28,0,0,
    70,55,114,120,60,28,0,0,0,
    16,88,124,56,28,0
1004 DATA56,126,60,32,6,15,2,6,10,84
    ,4,161,34,136,133,40,234,5
    6,239,78,98,198,48,96
1006 DATA0,0,0,0,0,114,127,114,0,0,0
    ,0,0,7,7,7,63,83,143,143,1
    37,174,254,174
1008 DATA63,83,143,143,137,142,142,1
    42,255,255,255,255,255,72,
    75,72,0,0,0,0,0,0,127,159
1010 DATA142,137,143,143,83,63,255,6
    3,78,24,255,255,255,255,25
    5,231,8,28,8,8,28,28,28,28
1012 DATA28,28,28,28,8,8,28,8,0,0,11
    4,127,114,0,0,0,0,0,78,254
    ,78,0,0,0
1014 DATA0,6,6,24,56,112,32,0,0,32,1
    12,56,24,6,6,0,0,96,96,24,
    28,14,4,0
1016 DATA0,4,14,28,24,96,96,0,63,83,
    143,143,137,190,190,190
1018 DATA9,12,10,11,0,25,0,12
```

ZAP!!

DUB SCROGGIN

ZAP!! demonstrates an important technique for fast animation – POKE graphics.

“ZAP!!” is an exciting and challenging VIC-20 game program designed for up to six players and up to five rounds per player. Each player may select from any of five skill levels and may change levels each round, if desired. Using keyboard controls, players maneuver a block around the screen and through a field of randomly placed and color coded graphic figures. The object is to run over and erase as many of these figures as possible in two minutes, but also to avoid running into asterisks and being zapped. After the player block is moved it cannot be stopped, but the direction of movement may be changed. The higher the skill level, the faster the block moves and the more asterisks there are. The number of scoring figures is increased so that a higher score is possible too.

The figures on the screen count differently toward the score. If a player is “zapped,” he retains his score, but his time is over. Players may run off the screen, but may strike a hidden asterisk if they do. A vertical wraparound feature prevents players from wandering too far off the screen. A variety of colors, graphics, and sound effects adds excitement to the program.

As with most computer games, proficiency at Zap!! will take some practice and a lot of concentration.

The player block is moved around the screen by the computer PEEKing at the keyboard to determine the value of the last control key pressed. A direction factor is then assigned to the variable DR (steps 590-620). When moving left, DR is -1, right is 1, up is -22 and down is 22. This factor is then added to the position of the block (B) (step 650). The old block is then erased by POKEing it to 32 (blank) and a new one is placed in position (step 570). This all happens so quickly that the illusion of motion is created.

Scoring and zaps are determined by PEEKs at the block’s position to see if any other figure is there (steps 670-720). Depending on the figure found at “PEEK (B)”, a score is assigned and the loop continues, or if the figure is an asterisk, a “Zap!!” routine is initiated and the round ends.

CHAPTER TWO

In each pass through the loop (steps 550-780), several things happen or are checked for. The elapsed time is printed, and there is a check to see if the time is up. If so, the loop is terminated and the round ends. A block is POKEd into position B. Steps 580 and 585 provide the vertical wraparound effect. A check is made for direction change input from the keyboard. A tone is sounded based on the current direction of movement. The old position of the block is erased and a new position is calculated. A check is made to see if any figures have been struck. If so, they are either scored or, in the case of an asterisk, the loop is terminated. After a new total score is calculated and displayed, the loop begins again.

Steps 640 and 760 are time delay steps to slow the block's motion and to increase speed as the skill level increases. If a faster or slower movement is desired, these steps may be altered.

A number of REMarks have been included in the program listing as an aid to understanding it, but I recommend that they not be typed on your computer. This program uses all but about 250 bytes of standard VIC-20 memory, and including all the remarks may result in an "out of memory" error.

```
10 PRINT"{CLEAR}"
20 DIM PL(6),R(5)
30 FOR Y=1 TO 5:FOR X=1 TO 6:Z(X,Y)=0:NEXT X:NE
   XTY
40 C=30720:TB=0:TS=0
50 POKE 36879,239
60 CP=0:GOTO 810
70 PRINTTAB(3)"{06 DOWN}BY DUB SCROGGIN"

80 REM-404 WOODROW ST.,FT. WALTON BEACH,
   FL 32548
90 CP=1
100 FORT=1 TO 2000:NEXT T
110 PRINT"{CLEAR}"
120 PRINTTAB(5)"{DOWN}DIRECTIONS"
130 PRINTTAB(5)"7777777777"
140 PRINT"{DOWN}YOU WILL HAVE 2 MIN.":PRI
   NT"TO GET YOUR BEST SCORE"
150 PRINT"{YEL}MOVEMENT:":PRINT"{DOWN}CRS
   R DN=LEFT":PRINT"CRSR RT=RIGHT":
   PRINT"F5=UP"
```

```

160 PRINT"F7=DOWN":PRINT"{HOME}{04 DOWN}"
170 PRINTTAB(14)"{03 DOWN}{WHT}SCORING:"
180 PRINTTAB(14)"{BLK}W=1"
190 PRINTTAB(14)"{CYN}Q=2"
200 PRINTTAB(14)"{YEL}Z=3"
210 PRINTTAB(14)"{RED}S=5"
220 PRINTTAB(14)"{WHT}A=10"
230 PRINT"{DOWN}YOU ARE: {BLU}{REV} {OFF}
"
240 PRINT"{DOWN}DON'T HIT A {PUR}*{BLU} O
R":PRINT"YOU WILL GET {PUR}ZAPPE
D."
250 PRINT"{WHT}{DOWN}PRESS ANY KEY TO STA
RT"
260 GETA$:IFA$=""THEN260
270 PRINT"{CLEAR}{WHT}HOW MANY ROUNDS (1-
5)"
280 INPUTRN:IFRN<1ORRN>5THENPRINT"HUH?":G
OTO270
290 PRINT"{DOWN}HOW MANY PLAYERS":PRINT"(
1-6)";
300 INPUTPN:IFPN<1ORPN>6THENPRINT"HUH?":G
OTO290
310 FORR=1TORN
320 FORP=1TOPN:PRINT"{BLU}{DOWN}PLAYER #"
;P
330 PRINT"{DOWN}WHAT SKILL LEVEL?"
340 PRINT"PRESS 0,1,2,3 OR 4";
350 INPUT S
360 IFS>4 ORS<0THENPRINT"HUH?":GOTO340
370 PRINT"{CLEAR}{BLU}{REV}SCORE TO BEAT:
";TB:PRINT"{REV}SKILL LEVEL:";SL
380 PRINT"{REV}PLAYER #";PB
390 FORT=1TO2000:NEXTT:PRINT"{CLEAR}"
400 DEF FN A(L)=INT(RND(1)* L )+7702
410 FORF=1TO40-2*S:D=FNA(483)
420 POKED,87:POKED+C,0:NEXTF
430 FORF=1TO25:D=FNA(483)
440 POKED,81:POKED+C,3:NEXTF
450 FORF=1TO10+4*S:D=FNA(505)

```

CHAPTER TWO

```
460 POKED,42:POKED+C,4:NEXTF
470 FORF=1TO19:D=FNA(483)
480 POKED,90:POKED+C,7:NEXTF
490 FORF=1TO14:D=FNA(483)
500 POKED,83:POKED+C,2:NEXTF
510 FORF=1TO9+S:D=FNA(505)
520 POKED,65:POKED+C,1:NEXTF
530 B=7932
540 TI$="0000000"
550 PRINT"{HOME}TIME: ";120-INT(TI/60);"{L
    EFT}  "
560 IFTI/60>=120THENGOTO930
570 POKEB,160:POKEB+C,6
580 IFB<7636THENB=8229+B-7635
585 IFB>8229THENB=7636+B-8230
590 IFPEEK(197)=31THENH=190:DR=-1:GOTO630
600 IFPEEK(197)=23THENH=200:DR=1:GOTO630
610 IFPEEK(197)=55THENH=210:DR=-22:GOTO63
    0
620 IFPEEK(197)=63THENH=220:DR=22
630 POKE36878,15:POKE36876,H
640 FORT=1TO30-5*S:NEXTT
650 POKEB,32:B=B+DR
660 SC=0
670 IFPEEK(B)=42THENGOTO790
680 IFPEEK(B)=87THENSC=1:GOTO740
690 IFPEEK(B)=81THENSC=2:GOTO740
700 IFPEEK(B)=90THENSC=3:GOTO740
710 IFPEEK(B)=83THENSC=5:GOTO740
720 IFPEEK(B)=65THENSC=10:GOTO740
730 GOTO760
740 TS=TS+SC
750 POKE36878,15:POKE36876,160+PEEK(B)
760 FORT=1TO30-5*S:NEXTT
770 PRINT"{HOME}{DOWN}SCORE=";TS
780 GOTO550
790 POKE36878,15
800 FORPI=1TO40:POKE36876,180-PI:NEXTPI
810 PRINT"{PUR}{RIGHT}{06 DOWN}&&&& &&&& ~
    &&&& && &&"
820 PRINT"{RIGHT}&&&& &&&& &&&& && &&"
```

```

830 PRINT"{RIGHT}  &  &  &  &  &  &&  &&"
840 PRINT"{RIGHT}  &  &&&&  &&&&  &&  &&"
850 PRINT"{RIGHT}  &  &&&&  &&&&  &&  &&"
860 PRINT"{RIGHT}&&&&  &  &  &"
870 PRINT"{RIGHT}&&&&  &  &  &  &&  &&{BLU
  }"
880 IFCP=0THEN70
890 POKE36878,0:POKE36876,0
900 FORT=1TO2000:NEXTT:PRINT"{CLEAR}"
910 PRINT"{WHT}{DOWN}{REV}YOU LASTED";INT
  (TI/60)-3;"{LEFT} SECONDS{OFF}"
920 GOTO970
930 POKE36878,15:FORAC=1TO80:POKE36876,21
  0-AC:NEXTAC
940 POKE36876,0:POKE36878,0
950 PRINT"{CLEAR}"
960 PRINT"{HOME}{04 DOWN} {BLU}{REV}....T
  IME IS UP...."
970 PRINTTAB(6)"{DOWN}{REV}{WHT}SCORE=";T
  S
980 Z(P,R)=Z(P,R-1)+TS
990 IFTS>TBTHENTB=TS:SL=S:PB=P
1000 PRINT"{DOWN}{BLK}ROUND #:";R;"{DOWN}"

1010 FORX=1TOPN
1020 PRINT"{YEL}PLAYER #";X;": ";Z(X,R)
1030 NEXTX
1040 TS=0:DR=0:H=0:PRINT"{DOWN}"
1050 NEXTP:NEXTR
1060 END

```


STARFIGHT3

DAVID R. MIZNER

This is an entertaining action game, but type it in carefully and SAVE it to tape before you RUN it. If you made a mistake, reset the VIC and LOAD the program, correct it, SAVE it, then RUN it.

STARFIGHT3 is a program that will let you fight off Klingons to save the Federation. Before you start typing away, a little word of warning is needed. This program *loves* memory. In fact, STARFIGHT3 will use it all up; so be careful entering the program. An extra space added now may cause a "no memory" message later.

Have fun!!!

Program Description

A new Galaxy is generated each time the program is RUN. Random numbers of stars (maximum of 25) and Klingons (maximum of three) are generated and, along with the Enterprise, are randomly placed in a 10x10 Galaxy.

The Enterprise is equipped with three photon torpedos for every Klingon, and three shield units. Three hits on the Enterprise from Klingon attacks will deplete its shield; a fourth hit will destroy the Enterprise. There will be self-destruction if the Enterprise runs into a star or a Klingon while traveling around the Galaxy.

Klingons (all that have not been destroyed) will fire at the Enterprise if your response time for a command is too slow or if your torp misses. Only one hit on the Enterprise is allowed per attack. Take note that the Klingons fire their torps in eight directions while the good guys can only fire in one direction at a time. However, neither side can fire through a star.

The stars and Klingons remain stationary throughout the game.

Program Directions

1. Observe operating procedures for VIC-20.
2. Commands
 - a. Move: VIC will request direction and distance. Direction is a number from 1 through 8, while distance is the number of spaces you want to move.
 - b. Torp: VIC will request a direction. A torp does not have a

fixed distance since a photon torpedo will travel until it hits a star, a Klingon, or a Galaxy boundary.

c. End: This command ends the game. "You surrendered" is the real meaning of "end."

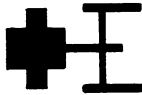
3. Scan

a. A scan is generated before each command request.

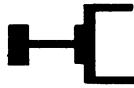
b. The Galaxy is displayed so you can see the actual location of stars, Klingons, and the Enterprise. At the same time, the direction code is printed out.

c. Scan code.

Enterprise



Klingon

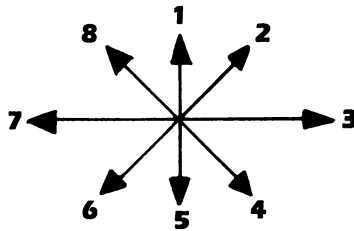


Star



4. Direction

The direction for moving the Enterprise or firing a photon torpedo is given by entering a number from 1 through 8. These numbers will let you move or fire a torp every 45 degrees.



5. Changing the game's difficulty

a. You can change the number of torps allowed by modifying line 180.

b. Another way is to change the time you are allowed before the Klingons fire. The value of TIS is changed by modifying lines 410, 500 and/or 1300.

To use STARFIGHT3 on a VIC with expanded memory, delete lines 30 through 100, substitute an "E" for the "@" in line 920 and substitute a "K" for the "#" in line 980. There will be no little ships now, but the program will run.

If you are receiving an OUT OF MEMORY error message, it is probably due to lines 30-100. Once the program has been RUN, these lines set aside memory that cannot be touched. So, if you

CHAPTER TWO

make a typo and try to rerun the program, you will run out of memory. The following procedure will let you make corrections:

1. Make the correction. 2. SAVE the program. 3. Turn the VIC off, then back on. 4. LOAD the corrected program. 5. RUN.

```
10 PRINT"{CLEAR}  ** STARFIGHT3 **"
20 PRINT:PRINT"DAVID R MIZNER,SEP81"
30 X=PEEK(56)-2:POKE52,X:POKE56,X:POKE51
   ,PEEK(55):CLR
40 CS=256*PEEK(52)+PEEK(51)
50 FORI=CSTQCS+511:POKEI,PEEK(I+32768-CS
   ):NEXT
60 FORI=7168TO7175:READJ:POKEI,J:NEXT
70 DATA15,68,228,254,228,68,15,0
80 FORI=7448TO7455:READJ:POKEI,J:NEXT
90 DATA7,12,204,252,204,12,7,0
100 POKE36869,255
110 DIMA%(10,10),KL(6)
120 FORI=1TO10
130 FORJ=1TO10
140 A%(I,J)=0
150 NEXTJ
160 NEXTI
170 K=INT(RND(1)*3+1):S=INT(RND(1)*25+1)
180 KC=K:T=3*K:H=3
190 FORI=1TOS
200 GOSUB840
210 IFA%(C1,C2)<>0THEN200
220 A%(C1,C2)=1
230 NEXTI
240 FORI=1TOK
250 GOSUB840
260 IFA%(C1,C2)<>0THEN250
270 A%(C1,C2)=2:KL(I)=C1:KL(I+3)=C2
280 NEXTI
290 GOSUB840
300 IFA%(C1,C2)<>0THEN290
310 A%(C1,C2)=3:E1=C1:E2=C2
320 PRINT:PRINT:PRINT"KLINGONS",K
330 PRINT:PRINT"TORPS",T
340 PRINT:PRINT"STARS",S
350 FORI=1TO3000:NEXT
360 GOSUB860
370 PRINT:PRINT"ENTER YOUR COMMAND"
380 PRINT"1=MOVE 2=TORP 3=END"
```

```
390 TI$="000000"
400 INPUTC
410 IFTI$<"000015"THEN440
420 GOSUB1130
430 GOTO360
440 ONCGOTO470,580
450 PRINT">YOU SURRENDERED"
460 GOTO1420
470 PRINT:PRINT"ENTER DIRECTION,DISTANCE"

480 C1=E1:C2=E2:TI$="000000"
490 INPUTC,D
500 IFTI$<"000015"THEN530
510 GOSUB1130
520 GOTO350
530 IFC>8ORD>14THEN490
540 A%(E1,E2)=0:GOSUB670
550 E1=T1:E2=T2
560 IFA%(E1,E2)=1ORA%(E1,E2)=2THENPRINT">
    HIT A STAR OR KLINGON":GOTO1420
570 A%(E1,E2)=3:GOTO360
580 IFT>0THENGOSUB1270
590 IFT>0ANDKC>K0THEN360
600 PRINT">NO MORE TORPS"
610 IFKC>1THEN640
620 PRINT">RAM LAST KLINGON"
630 GOTO470
640 PRINT">YOU'RE OUTNUMBERED"
650 PRINT">FEDERATION IS LOST"
660 GOTO1420
670 ONCGOTO690,700,710,720,730,740,750
680 U=-1:V=-1:GOTO760
690 U=-1:V=0:GOTO760
700 U=-1:V=1:GOTO760
710 U=0:V=1:GOTO760
720 U=1:V=1:GOTO760
730 U=1:V=0:GOTO760
740 U=1:V=-1:GOTO760
750 U=0:V=-1
760 FORI=1TOD
770 T1=C1+I*U:T2=C2+I*V
780 IFT1<1ORT1>10ORT2<1ORT2>10THEN820
790 IFA%(T1,T2)>0THEN830
800 NEXTI
810 GOTO830
820 T1=C1+(I-1)*U:T2=C2+(I-1)*V
830 RETURN
840 C1=INT(RND(1)*10+1):C2=INT(RND(1)*10+1)
```

CHAPTER TWO

```
850 RETURN
860 PRINT:PRINT" *** SCAN ***"
870 PRINT:PRINT" ++++++++"
880 FORI=1TO10
890 PRINT" +";
900 FORJ=1TO10
910 ONA%(I,J)+1GOTO940,960,980
920 PRINT"@";
930 GOTO990
940 PRINT" ";
950 GOTO990
960 PRINT"*";
970 GOTO990
980 PRINT"#";
990 NEXTJ
1000 ONIGOTO1020,1030,1040,1050,1060,1070,
    1080
1010 GOTO1090
1020 PRINT"+ COURSE":GOTO1100
1030 PRINT"+":GOTO1100
1040 PRINT"+ 1":GOTO1100
1050 PRINT"+ 8 2":GOTO1100
1060 PRINT"+ 7 3":GOTO1100
1070 PRINT"+ 6 4":GOTO1100
1080 PRINT"+ 5":GOTO1100
1090 PRINT"+"
1100 NEXTI
1110 PRINT" ++++++++"
1120 RETURN
1130 FORM=1TOK
1140 C1=KL(M):C2=KL(M+3):D=14
1150 IFA%(C1,C2)=0THEN1210
1160 PRINT">KLINGON SHOOTING"
1170 FORC=1TO8
1180 GOSUB670
1190 IFA%(T1,T2)=3THEN1230
1200 NEXTC
1210 NEXTM
1220 GOTO1260
1230 H=H-1:IFH<0THEN650
1240 PRINT:PRINT">ENTERPRISE IS HIT"
1250 PRINTH"SHIELD UNITS LEFT"
1260 RETURN
1270 PRINT:PRINT"PHOTON TORP DIRECTION"
1280 TI$="000000"
1290 INPUTC
1300 IFTI$<"000015"THEN1330
1310 GOSUB1130
```

```
1320 GOTO1410
1330 C1=E1:C2=E2:T=T-1:D=14
1340 IFC>8THEN1270
1350 GOSUB670
1360 IFA%(T1,T2)<>2THEN1400
1370 A%(T1,T2)=0:KC=KC-1
1380 IFKC=0THENPRINT"> FEDERATION SAVED <"
      :GOTO1420
1390 GOTO1410
1400 GOSUB1130
1410 RETURN
1420 PRINT:PRINT
1430 INPUT"ANOTHER GAME 1=YES";Z
1440 IFZ=1THEN120
1450 END
```

Alphabetizer

JIM WILCOX

This simple bubble sort can be used in many programs which need an alphabetizing sort. Lines 10-20 enter the array A\$; lines 80-120 sort A\$, when they are given VAR, the number of variables.

The following program will alphabetize letters, or put numbers in order from lowest to highest. The first thing that will happen is that the screen will clear and the message "HOW MANY VARIABLES?" will appear on the screen. You then type in the number of names you wish to sort. The variable "VAR" will take on the value typed in. The statement at line number 20 will set the amount of variables of the dimensioned variable "A\$". If you are stuck on dimensioned variables, read on.

DIMensioned Variables

DIMensioned variables can be compared to houses on a street. Let's say the house numbers on this block start at 1 and end at 10. They all belong to the street named, say, "Washington." To make things easier than naming each house after a different president, they are given numbers. There might be another house with the number 2, but not on the same street. The name of the street is the variable, but there is more than one house on the street, and houses are variables too. To get a letter to house #2 on Washington Street, one would have to write the person's name, "Jones," who would reside at 2 Washington Street. In a computer program, one could set the variable WASHINGTON\$(2) = "JONES". 1 Washington Street might have the "Georges" living there, so the variable would be WASHINGTON\$(1) = "GEORGE". So a DIMensioned variable is a variable that has other variables related to it; i.e., all the people on the block have in common the fact they live on Washington Street.

I recommend that you try a small list first, such as ten of the letters of the alphabet mixed up. It will not take long to put the characters in order, and the programmer can tell whether the program was typed in properly. On longer lists it becomes tempting to hit the RUN/STOP key to see if the computer is stuck in an endless loop, but the longer the list, the longer it takes.

```
10 INPUT "{CLEAR}HOW MANY VARIABLES";VAR
20 DIM A$(VAR+22)
30 FOR A=1 TO VAR
40 PRINT "#";A;
50 INPUT A$(A)
60 NEXT A
70 PRINT"ALPHABETIZING"
80 FOR A=1 TO VAR-1
90 FOR B=A+1 TO VAR
100 IF A$(B)<=A$(A)THENSMS=A$(B):A$(B)=A$(A):A
    $(A)=SMS
110 NEXT B
120 NEXT A
130 PRINT"FINISHED ALPHABETIZING"
140 POKE 36878,8
150 POKE 36874,250
160 FOR A=1 TO 500
170 NEXT A
180 POKE 36878,0
190 POKE36874,0
200 FOR A=1 TO VAR STEP 22
210 FOR B=A TO A+21
220 PRINT A$(B)
230 NEXT B
240 GET A$:IF A$=""THEN240
250 NEXT A
260 END
```


Count The Hearts

CHRISTOPHER J. FLYNN

As yet there are very few computer programs suitable for young children. However, the computer can be a fascinating learning aid for children. Graphics and sound will hold the attention of the very young while teaching. "Count the Hearts" is a colorful program for preschoolers.

"Count the Hearts" is a program which will help you develop your child's counting skills. VIC will display a certain number of hearts on your television screen. Ask your child to count them. If your child can correctly count the hearts, he or she will be rewarded by a duet of chirping birdies. To challenge older children, you can limit the time VIC allows for a response.

Once it is set up, no reading is required to play "Count the Hearts." However, preschoolers will probably need you to help them with the keyboard.

Setting Up

When you first start "Count the Hearts," it will ask you for a range of numbers and a time limit.

You can tailor the game to your child's counting skills by trying different number ranges. For example, you may want to start with numbers between one and five. Gradually, a child will work up to counting up to ten. If you notice difficulties with some numbers, you might want to work within that range (say from six to nine).

Here is how VIC will ask you to set the number range:

1. VIC will display:
ENTER NUMBER RANGE
LOW NUMBER (1) ?
2. You should type in the low number in the range (don't forget to hit RETURN). If you just hit RETURN, VIC will use one as the low number.
3. Next VIC will ask:
HIGH NUMBER (9)?
4. Now type in the high number. Again, if you just hit RETURN,

VIC will use nine as the high number.

VIC will make sure that your low number is really lower than your high number. It will also make sure that neither number is greater than 484. Why 484? Well, that's how many spaces are left on the screen for displaying the hearts.

The time limit gives you a way to speed up "Count the Hearts." If you don't take a guess at how many hearts there are within the time limit, then VIC will let you know that time's up. VIC will then just start another game.

VIC will ask you for the time limit:

5. VIC will display:

TIME LIMIT PER SET

SECONDS (120) ?

6. Type in the number of seconds you want to use for the time limit. If you just type RETURN, VIC will set the time limit to 120 seconds, or two minutes.

By the way, if, in any of the above steps, VIC didn't understand your response, it will either ask the question again or ask you to repeat your response.

Counting Hearts

The television screen will go blank for just an instant. In that brief instant VIC is deciding how many hearts it will ask you to count. Then, one by one, VIC displays the hearts at random locations on the television screen. As it shows each new heart, VIC says in a deep voice, "BEEP!" Notice how VIC paints the hearts in different colors.

Now VIC will ask:

HOW MANY HEARTS?

Ask your child to count them. Type in the number (don't forget RETURN!) and see what happens. What happens if your child gives the right answer? How about a wrong answer? What is your child's reaction?

VIC will start a new game when the right answer is typed in or when time runs out and nothing has been heard from the keyboard. VIC is very patient with small folks learning to count. When a wrong answer is given, VIC just resets its timer and gives them another try.

Scoring

When finished playing "Count the Hearts," just hit the F1 key in reply to the "HOW MANY HEARTS?" question. VIC will promptly

CHAPTER TWO

clear the screen and tell you:

- how many games were played
- how many correct answers there were
- how many wrong answers there were
- how many times the player ran out of time

By keeping track of the number range (VIC shows you the range you used) and the scores, you can note your child's progress. For example, do you notice a little slowness in your child's learning to count past ten? We did. That seems to be the upper limit for our three year old for a while.

Hints

You probably don't need to be reminded that the attention span of preschoolers is not long. Try to move on to another activity before your child gets bored and begins to act silly. You want your child to remember counting as something that is fun to do.

One way for you to help beginners is for you to point to the hearts very slowly, one by one. Let your child count them as you point to them. Gradually your child will take over the pointing. And, before you know it, your child will be typing in the numbers on the keyboard! Experiment. Try out different arrangements. What works best for you?

The program will run on a standard VIC without memory expansion. If you need to, you should be able to modify "Count the Hearts" without too much trouble.

Now you're ready to play "Count the Hearts"! But remember, to stop the game and see your score, all you need to do is press the F1 key. Have fun.

```
100 REM VIC-20
110 REM COUNT THE HEARTS
120 REM V1.0 7/81
130 REM COPYRIGHT 1981 HOMESPUN SOFTWARE
200 REM
210 REM HEARTS.BEGIN
220 GOSUB 30000
230 REM PLAY GAMES
240 GOSUB 1000
250 IF Q=0 THEN 240
260 REM HEARTS.END
270 GOSUB 31000
280 END
```

```
1000 REM PLAY GAMES
1010 PRINT CHR$(147)
1020 REM DISPLAY HEARTS
1030 N=LO+INT((HI-LO+1)*RND(1))
1040 FOR I=1 TO N
1050 P=INT(484*RND(1))
1060 CL=INT(8*RND(1)):IF CL=1 THEN 1060
1070 IF PEEK(VA+P)=83 THEN 1050
1080 POKE VA+P,83
1090 POKE CA+P,CL
1100 POKE VL,15
1110 POKE S2,200
1120 FOR Z=1 TO 400:NEXT
1130 POKE S2,0:POKE VL,0
1140 NEXT I
1150 G=G+1:REM GAMES
1160 PRINT CHR$(19);
1170 FOR I=1 TO 21:PRINT " ";:NEXT
1180 PRINT CHR$(19);"HOW MANY HEARTS ? ";
1190 REM GET RESPONSE
1200 GOSUB 3000
1210 IF R$="QUIT" THEN Q=1:RETURN
1220 IF R$="TIME" THEN GOSUB 9000:RETURN
1230 REM O.K.?
1240 R=VAL(R$)
1250 IF R<>N THEN GOSUB 5000:GOTO 1160
1260 IF R=N THEN GOSUB 7000
1270 RETURN
3000 REM TIMED RESPONSE
3010 T1=TI+SC*60
3020 R$=""
3030 REM TRY A KEY
3040 GET A$
3050 IF TI>T1 THEN R$="TIME":RETURN
3060 IF A$="" THEN 3040
3070 IF ASC(A$)=133 THEN R$="QUIT":RETURN
3080 IF ASC(A$)=13 THEN RETURN
3090 IF ASC(A$)=20 AND LEN(R$)>0 THEN GOSUB 330
    0:R$=LEFT$(R$,(LEN(R$)-1)):GOTO 3040
3095 IF ASC(A$)=20 THEN 3040
3100 PRINT A$;
3110 IF A$<"0" OR A$>"9" THEN GOSUB 3300:GOTO 3
    040
3120 R$=R$+A$
3130 GOTO 3040
3300 REM BACKSPACE
```

CHAPTER TWO

```
3310 PRINT CHR$(157);
3320 PRINT " ";
3330 PRINT CHR$(157);
3340 RETURN
5000 REM WRONG
5010 WR=WR+1
5030 REM UFO-VARIATION
5040 POKE VL,15
5050 FOR L=1 TO 15
5060 POKE SB,42
5070 FOR M=200 TO 220+L*2
5080 POKE S3,M
5090 NEXT M
5100 POKE SB,25
5110 FOR Z=1 TO 25:NEXT Z
5120 NEXT L
5130 POKE VL,0:POKE S3,0
5140 POKE SB,27
5150 RETURN
7000 REM RIGHT
7010 RI=RI+1
7020 REM BIRDS VARIATION
7025 PRINT CHR$(19);:FOR Z=1 TO 21:PRINT " ";:N
    EXT Z
7030 POKE VL,15
7040 FOR L=1 TO 20
7050 PRINT CHR$(19);SPC(5);CHR$(106);CHR$(113);
    CHR$(107);
7055 PRINT SPC(5);CHR$(117);CHR$(113);CHR$(105)
    ;
7060 FOR M=254 TO 240+RND(1)*10 STEP -1
7070 POKE S3,M
7080 NEXT M
7090 POKE S3,0
7100 FOR M=1 TO 100:NEXT M
7110 PRINT CHR$(19);SPC(5);CHR$(117);CHR$(113);
    CHR$(105);
7115 PRINT SPC(5);CHR$(106);CHR$(113);CHR$(107)
    ;
7120 FOR M=1 TO 120*RND(1):NEXT M
7130 NEXT L
7140 POKE S3,0:POKE VL,0
7150 RETURN
9000 REM TIME
9010 TM=TM+1
9020 VM=VA+253:CM=CA+253
```

```
9030 REM TONE
9040 POKE S3,240:POKE VL,15
9050 J=0
9060 FOR L=15 TO 0 STEP -2
9070 POKE VM+J,81:POKE CM+J,2
9080 POKE VM-J,81:POKE CM-J,2
9090 POKE VM+22*J,81:POKE CM+22*J,2
9100 POKE VM-22*J,81:POKE CM-22*J,2
9110 FOR Z=1 TO 50:NEXT Z
9120 POKE VM+J,32
9130 POKE VM-J,32
9140 POKE VM+22*J,32
9150 POKE VM-22*J,32
9160 FOR Z=1 TO 50:NEXT Z
9170 POKE VL,L
9180 J=J+1
9190 NEXT L
9200 POKE VM-2,20:POKE CM-2,4
9210 POKE VM-1,9:POKE CM-1,4
9220 POKE VM,13:POKE CM,4
9230 POKE VM+1,5:POKE CM+1,4
9240 POKE S3,0:POKE VL,0
9250 FOR Z=1 TO 2000:NEXT Z
9260 RETURN
30000 REM HEARTS.BEGIN
30010 REM CONSTANTS/VARS
30020 VA=7702
30030 CA=38422
30040 SB=36879
30050 VL=36878
30060 S2=36875
30070 S3=36876
30080 S4=36877
30090 LO=1
30100 HI=9
30110 SC=120
30120 G=0
30130 TM=0
30140 RI=0
30150 WR=0
30160 Z=RND(-TI)
30170 PRINT CHR$(147);
30180 PRINT SPC(8);"VIC-20"
30190 PRINT
30200 PRINT"      COUNT THE HEARTS"
30210 PRINT:PRINT
```

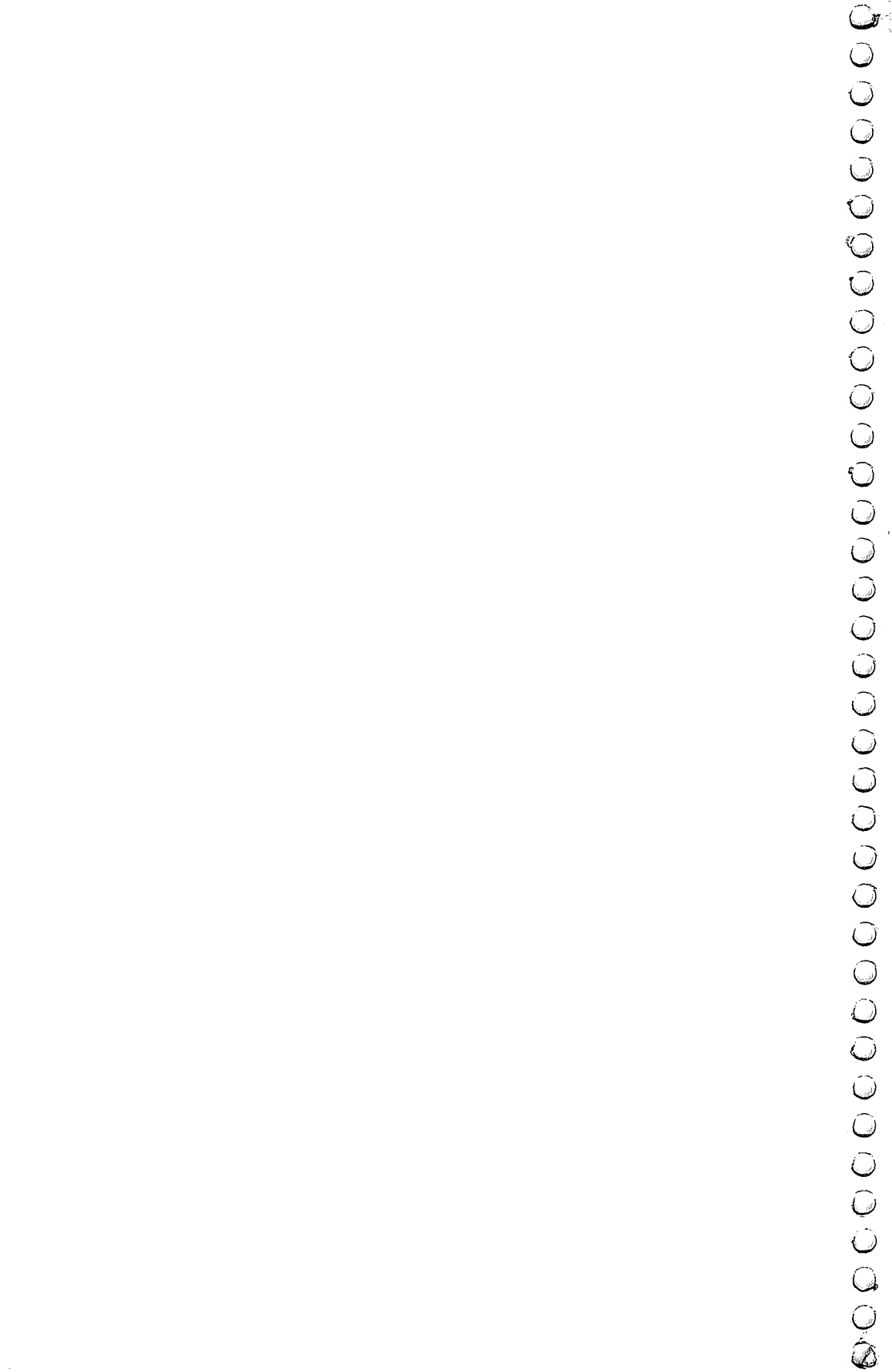
CHAPTER TWO

```
30215 PRINT CHR$(158);
30220 PRINT"    COPYRIGHT 1981"
30230 PRINT"    HOMESPUN  SOFTWARE"
30235 PRINT CHR$(31);
30240 PRINT:PRINT
30250 PRINT"ENTER NUMBER RANGE"
30260 INPUT"LOW NUMBER (1)";LO
30270 LO=ABS(INT(LO))
30275 IF LO<1 OR LO>484 THEN PRINT "SORRY":LO=1:
      GOTO 30260
30280 INPUT"HIGH NUMBER (9)";HI
30290 HI=ABS(INT(HI))
30300 IF HI<=LO OR HI>484 THEN PRINT"SORRY":HI=9
      :GOTO 30260
30310 PRINT
30320 PRINT"TIME LIMIT PER SET:"
30330 INPUT"SECONDS (120)";SC
30340 SC=ABS(INT(SC))
30345 IF SC<1 THEN PRINT "SORRY":SC=120:GOTO 303
      30
30350 PRINT:PRINT
30360 PRINT"KEY F1 TO STOP"
30370 PRINT:PRINT
30380 PRINT"THANK YOU. HAVE FUN!"
30390 FOR Z=1TO2000:NEXT
30400 RETURN
31000 REM HEARTS.END
31010 PRINT CHR$(147)
31020 PRINT"COUNT THE HEARTS"
31030 PRINT:PRINT
31032 PRINT"LOW #",LO
31034 PRINT"HIGH #",HI
31036 PRINT:PRINT
31040 PRINT "# GAMES",G
31050 PRINT "# RIGHT",RI
31060 PRINT "# WRONG",WR
31070 PRINT "# TIME OUTS";TM
31080 RETURN
```

CHAPTER THREE

PROGRAMMING TECHNIQUES





PRINTing With Style

JAMES P. McCALLISTER

This extensive tutorial covers nearly everything you'll need to know about PRINT.

One of the beauties of VIC BASIC is that simple commands are all you need to print information on the screen. As a novice, you can concentrate on the parts of the program you're most interested in, and not have to cope with numerous rules devoted to the output itself.

Sooner or later, though, we all start to get fussy about the appearance of the screen. And it is possible to create very effective printed displays on the screen. Given the limited format – 22 characters wide by 23 lines high – the key is not to waste any of that precious area. Naturally a certain amount of open space is needed to avoid a cluttered look. And you've got to put the printing *exactly* where you want it, not one column to the left or right. Unused spaces at the right and left edges are wasted unless they serve a purpose, and unintentional line skipping shouldn't be tolerated.

This article will show you how easy it is to achieve these goals. We'll end up with a program that simulates an Electronic Checkbook and makes use of the entire screen.

The PRINT Statement – The Programmer's Paintbrush

If the screen is your canvas, the PRINT statement is your paintbrush. Basically, it prints the values of a list of variables on the screen. The list is flexible. Instead of actually naming a numeric variable in the list, you may give a function (formula) which in effect computes a variable. When the PRINT statement encounters the function in the list, it will compute its value and print it. Also, instead of actually naming a string (alpha-numeric) variable in the list, you can create a literal string (characters enclosed in quotation marks). The result of all this is still a list of variables whose values are to be printed. But in some cases you create the variables within the PRINT list, while in others you compute the variables earlier and then include them by name in the PRINT list. The list can be only one variable long, which doesn't even look much like a list. But it's called a list, just the same.

Just what the results of a PRINT statement *look* like depend on

two things: (1) the values of the variables in the list, and (2) the use of the four PRINT statement position controls. The four position controls are the comma (,), semicolon (;), SPC, and TAB. In the list, variables (or functions, or literals) are separated by one or more position controls. Each control affects *where* the next printing will take place. Thus, when the PRINT statement encounters a variable in the list, the next spot where printing will occur on the screen is already determined by what has gone before.

First, consider the variables. If the variable is a numeric variable, its value will be printed with either a minus sign or a space before, and one space after. The value will be exactly the one stored for that variable name, with no rounding. If there is no decimal fraction, no decimal point will be printed. If there is a decimal fraction, all trailing zeros on the right end will be deleted – the decimal fraction cannot end with 0.

If the variable is a string variable, the characters will be printed with no extra space before or after. Conversion of numeric variables to string variables is often helpful. If X is a numeric variable, STR\$(X) will print with either a minus sign or space to the left of X, but with no space to the right. If a string variable contains characters which have a non-printing function, such as CLR, cursor movement, color control, or the like, those functions will execute and affect the display accordingly, when the PRINT statement is executed. The cursor controls affect the next print position even though the cursor isn't visible on the screen.

When it comes to using these non-printing string variable characters to control print position, the sky's the limit! There's almost nothing you can't do with a little imagination and the cursor controls. You can make life a whole lot easier, though, by understanding the four PRINT statement position controls. Until you do, you will probably: (1) use many more cursor characters than you need; (2) use a lot of trial and error to get your display the way you want it; and (3) find your program display isn't consistent when run with a variety of data.

Using , ; SPC And TAB

The variables in the PRINT list *must* be separated by one or more of these PRINT statement position controls:

- , moves the next print position to column 11 or column 0 of the next line, whichever is next. Multiple commas move the next print position forward 11 spaces for each additional comma.
- ; specifies that the next print position will be the very next one

available. Remember, though, that a numeric variable comes with a built-in space after, and maybe one before. Multiple semicolons have the same effect as just one. You still get the next position available. A semicolon following another position control has no effect whatsoever, except to use up a byte of memory.

SPC(X) moves the next print position forward X spaces (255 maximum). X can be a number or a function.

TAB(X) moves the next print position forward to column X. If the next print position available is already higher than X, TAB is ignored. X can be a number or a function. "Multi-line" lines, which we'll talk about later, make TAB a little trickier than the others.

SPC and TAB separate the variables in a PRINT list just as well as a comma or semicolon and no additional punctuation is needed.

If desired, comma, SPC, or TAB can be used at the beginning of a PRINT list, before any variable, with the expected result. The semicolon has no effect at the beginning of a list.

At the end of a PRINT list, after the last variable, any of the four position controls can be used. The result is as if the list in the next PRINT statement encountered were a continuation of the list in the current PRINT statement. This is a most useful feature. Further, the *absence* of any of the four position controls at the end of a PRINT list is also an important control in itself. The result is that the next print position is moved to the beginning of a new line. An empty PRINT statement, with no list, is a special case of this rule. The next print position moves to the beginning of another line – even if the empty PRINT statement was already at the beginning of a new line.

"Multi-line" Lines

Ordinarily, a line is 22 columns wide – positions 0 through 21. But suppose you print a variable in the rightmost column, position 21. Then 22 more positions are automatically added to that line and the next print position is 22. Of course, position 22 is on the left edge of the screen, just like 0, but the label for that position is 22, not 0. If, for example, the next item in the list were TAB(9) it would be ignored, because position 9 is already past. TAB(31) would work just fine, though, and land on the intended spot. This process can continue with up to three continuation lines, going all the way to position 87.

The multi-line process happens automatically, and you don't have to pay much attention to it except for two reasons. First, if you want to use the TAB position control, you'll need to know which

position you're in. If you didn't print in column 21 of the last line, there's no question about it – the current line is positions 0 through 21. If the situation is complicated, you can find out where you are by using the function `POS(X)`, where `X` is a "don't care" number such as 0. This function returns the current next printing position. If you want to TAB to position 9 or its multi-line equivalent, you can use `TAB(9 + INT(POS(0)/22)*22)`. If the situation allows, it's much better to use the SPC position control, because you don't need to know where you are. Four spaces forward means the same thing, no matter what your position.

The second – and even more important – reason to think about "multi-line" lines is to eliminate unwanted line skipping after ending in the rightmost column. After all, with only 22 columns to start with, we don't want to shy away from column 21 just because it tends to force line skipping. When you print a variable that ends in position 21, simply put a semicolon after the variable (or function, etc.) in the list. This will always cause the next printing to be on the very next line, regardless of which print position (21, 43, 65 or 87) you were in. This works without any thought about the mechanics, but if you're curious, the Screen Line Link Table allows you to see what's going on. We'll look at that as a part of the program.

Using The Full Screen

You're almost ready to make full use of the VIC's screen format, if you'll keep in mind these points:

1. Remember the spaces that come "free" with every numeric variable – one before, unless the number is negative, and one after all numbers. The function `STR$(X)` gets rid of the one on the right. If you know the number isn't negative, `MID$(STR$(X),2)` eliminates the space on the left as well. For positive whole numbers from 0 to 99, `RIGHT$(STR$(X),2)` occupies just two positions and is right justified. That is, the right digits of every number in the column are aligned. To print numeric variables on the left or right edge of your screen, you'll have to use tricks like these.

2. Use a semicolon after ending a variable in column 21, to avoid skipping the next line. This also applies if the "free" space to the right of a numeric variable lands in column 21. (But the Screen Line Link Table turns out differently.)

And now, for the last frontier – the bottom of the screen. If your program completes execution, VIC will skip a line and print `READY`. If you have already printed on 23 lines, this will cause at least two lines at the top to roll off. It's easy enough to prevent this by inserting

XX GETA\$:IFA\$="" THEN XX (XX is the line number)

after the last PRINT statement. This forces VIC to keep executing this line until you press a key, which you won't do until you've finished looking at the display. But the lower right corner is still a holdout!

If you print in the lower right corner, VIC will immediately roll up another blank line (several, if a "multi-line" line is at the top) to be ready for the next PRINT list item. And thus you lose the top of your hard-won display. Simply not printing in the lower right corner may be the solution, in which case you're all set to use your screen. On the other hand, if you've been putting numbers all the way down the right side of the screen, you might have a strong desire to print the total on the bottom line, directly beneath. Some day Commodore may tell us a better way, but until then, insert this line:

IF PEEK(214)=22 THEN POKE 214,21

Location 214 is one of several which keep track of cursor and print line position, but it's the one that does the trick. It fools a portion of the operating system into thinking you're only on the second last line. This statement must come after the next print position has moved to the bottom line, but before you print in the lower right corner.

Electronic Checkbook Display

Program 1 provides the display for an Electronic Checkbook. The data is simulated (lines 40, 50, and 140) to keep the program short, but you can see the exciting possibilities with the PRINT techniques we've discussed.

The dates are the first item in the PRINT list of line 150. The RIGHTS\$(STR\$(DT),2) function eliminates a useless space to the left of the two-digit numbers, and also right-justifies the one-digit numbers for a neater appearance. The first SPC in line 170 right-justifies CK on column 5, regardless of the number of digits, one, two, or three. And don't miss line 180; that lets us print in the lower right corner.

The subroutine beginning at 995 prints the dollar amounts with trailing zeros in the cents columns. The cents conversion is in line 1020. The trick is to convert the cents to a rounded whole number, add 100 to it, and print only the two right hand digits. That's the only way to get the \$1500.00 in the first entry to keep all its zeros. Line 1030 computes the value for the multi-line TAB function in the PRINT statement.

Since subroutines can't pass variables as an argument, it's necessary to set the variables needed before calling the subroutine.

CHAPTER THREE

The dollar amount to be printed is put in X. Optionally, $D = 1$ causes a dollar sign to be printed, as in the first entry of the Checkbook, and $R = 1$ causes the amount to be printed in the reverse font. The subroutine resets D and R, so it's only necessary to use them as exceptions. Also, I couldn't resist printing the negative amounts in purple, because it's so easy to do.

The Program 1 add-on should be merged with the first program, after you've got Program 1 running. It produces a second page after you press the space bar or any other key. The program stores and then prints the values of the Screen Line Link Table. This table is found in memory locations 217-239. Each location corresponds to a line of the screen. For the top half, 158 designates a starting line and 30 shows a continuation line. For the lower half, the numbers are 159 and 31. The values were PEEKed while the Checkbook display was on the screen, and apply to it.

In the Checkbook display, line 0 of the display was a starting line. Since line 0 didn't print in column 21, line 1 was also a starting line. Lines 2 and 3 are continuation lines, but since line 3 didn't print in column 21, line 4 was a starting line, again. From then on, column 21 was used on every line. Therefore, the lines are grouped in fours – one starting line followed by three continuation lines. You'll never need to refer to this table to do your PRINTing, but sometimes understanding what's going on in VIC's brain can be helpful.

The add-on program uses simpler methods for printing than the main program. The only shortcoming is that the columns are left-justified while we would prefer right justification. For program analysis and the like, this is probably a good trade-off between PRINT statement complexity and results. Another note – observe the automatic line skip after printing the top line of the add-on display on the screen, because it is a full line. If you want to experiment, put a semicolon at the end of statement 220, and the skipped line will not be there.

In the first part of the program, line 90 initializes the random number generator so the results are always the same. You can delete line 90, and successive runs of the program will give changing dollar amounts.

Once you have a feel for how the program does the PRINTing, here's an exercise for the adventurous. The variable for whole dollars, DLS, has either a space or a minus sign to the left of the digits. You can see the spaces on the reverse image BALANCES in the display. The positive numbers might look better without the space. See if you can fix things so the minus sign comes through,

but there's no space before the positive numbers. It's not hard – good luck, and good PRINTing.

Program 1.

```

9 REM CONSTS FOR SUBR
10 C$(0)="" : C$(1)="{PUR}"
20 R$(0)="" : R$(1)="{REV}"
30 D$(0)="" : D$(1)="$"
35 REM CONSTS FOR MAIN PROG
40 PP$="PPPPPPP"
50 IT$="IIIII"
60 H1$="{CLEAR}      CHECKBOOK DEMO"
70 H2$="{REV}DATE   PAYEE      {OFF}AMOUNT"
80 H3$=" {REV}CK#    ITEM      BALANCE{OFF}"
85 REM STRT MAIN PRGM
90 X=RND(-1)
100 PRINTH1$,H2$,H3$;
110 PRINT" 1      BAL FWD"
120 BL=1500:X=BL:R=1:D=1:GOSUB1000
130 FORN=2TO10
140 DT=2*N:CK=N+N*N:AM=INT(5E+4*RND(1))/100:RE
    M SIMULATED DATA
150 PRINTRIGHT$(STR$(DT),2);SPC(4);PP$;
160 X=AM:GOSUB1000
170 PRINT SPC(5-LEN(STR$(CK)));STR$(CK);SPC(3)
    ;IT$;
180 IFPEEK(214)=22THENPOKE214,21
190 BL=BL-AM:X=BL:R=1:GOSUB1000:NEXT
200 GETA$:IFA$=""THEN200
995 REM $ PRINT SUBR
1000 C=0:IFSGN(X)=-1THENC=1
1010 IX=INT(ABS(X))*SGN(X):DL$=STR$(IX)
1020 DC$=RIGHT$(STR$(INT((1000*ABS(X-IX)+5)/10)
    +100),2)
1030 TB=22*INT(POS(0)/22)+19-LEN(DL$)-D
1040 PRINTTAB(TB)C$(C);R$(R);D$(D);DL$;". ";DC$;
    "{OFF}";"{BLU}";
1050 R=0:D=0:RETURN

```

Program 2. Add to Program 1.

```

5 DIM SC(22)

```


CHAPTER THREE

```
210 FOR N=0 TO 22:SC(N)=PEEK(217+N):NEXT
220 PRINT"{CLEAR}SCREEN LINE LINK TABLE"
230 PRINT"LINE CODE  LINE CODE",,,
240 FORN=0TO10
250 PRINT N;TAB(4);SC(N),N+12;SC(N+12):NEXT
260 PRINT" 11 ";SC(11)
270 END
```

Train Your PET To Run VIC Programs

LYLE JORDAN

The VIC, thanks to a built-in "relocating loader," can easily LOAD PET/CBM tapes. The PET/CBM can load programs written on a VIC, but it isn't possible to use these programs unless they are moved from memory location \$1001 to \$0401.

Have you already wished for the capability to renumber your VIC program, or to get a printout, or to save it on a disk?

This could be especially frustrating if you have a PET, a printer, and a disk sitting idly nearby.

Or how about satisfying the desire to "upload" your VIC program into a PET? This article will give you a couple of quick and easy ways to do just that.

The PET BASIC programs start to occupy memory at location 1025 decimal or \$0401 hex. For the VIC, programs will start at 4097 decimal or \$1001 hex. To make things compatible, start by putting a one line program into your PET (example: 1 REM). Now load your VIC program by typing "load". The VIC program will load just fine, but will have a starting location of \$1001 hex and if you do a LIST, it won't show up at all. You will see only your one line program, 1 REM.

To get to the VIC program, you will need to change the forward linking pointers. This can be accomplished by doing a SYS 54386 (to get into the machine language monitor) and then by changing two memory locations.

First look at memory locations \$0400 to \$0407, by typing:

M 0400,0407

The PET will display the following:

```
.M 0400,0407  
.: 0400 00 07 04 01 00 8F 00 00
```

Next list memory \$1000 to \$1007 and see:

```
.M 1000,1007  
.: 1000 AA 18 10 0A 00 99 22 56
```

CHAPTER THREE

This display will vary depending on the first line of your VIC program. My first line was `10 PRINT "VIC-20"`.

Now you can change the "07" and the "04" at locations \$0401 and \$0402. You want this to point to the location of the first forward pointer of the VIC program, so the "07" becomes "01" and the "04" is changed to "10". Make the changes, press RETURN, and cursor down to the last line displayed, type "x", and then press return again. When the PET gives the "READY", you are back in BASIC and can do a LIST. What appears is the one line, `1 REM`, followed by the VIC program.

Having served its purpose, line one can now be removed, and the VIC program will be copied into the normal start of PET BASIC at location 1025 decimal or \$0401 hex.

If you have The BASIC Programmer's Toolkit from Palo Alto IC's, this entire procedure can be replaced by simply activating the Toolkit, and typing "APPEND".

I will have a lot of use for both of these procedures. Some that come to mind immediately are such things as getting a VIC program listing on a PET printer, renumbering a VIC program, and compacting a program so as to make the best possible use of the VIC's 3.5K of memory. I hope that this simple procedure will prove useful to others.

User Input

WAYNE KOZUN

This tutorial on keyboard input takes you from INPUT to PEEK. Following the tips here, you can maximize the potential of VIC's full stroke keyboard.

There are three main ways to accept input from the keyboard during the running of a program. The two more popular methods are the INPUT and GET commands. The final and least popular method is to PEEK location 197 or 201.

INPUT

The INPUT is the easiest way to receive data from the user, and therefore the most popular. It assigns the characters entered to the given variable. To signal that you have finished entering, you must press RETURN, and the program resumes its execution.

Basically two different types of variables are used here. Floating point variables (such as X) are used to input numbers. String variables (such as XS) are used to input all kinds of characters. String variables are denoted by a letter followed by a dollar sign. Try this program to see the INPUT command in its easiest form.

```
10 PRINT"ENTER A NUMBER"
20 INPUT A
30 PRINT "ENTER A WORD"
40 INPUT A$
50 PRINT A,A$
60 END
```

The above program shows how to use the INPUT command with both string and floating point numeric variables. The two variables are entered in lines 20 and 40. The program halts when it comes to an INPUT, and it doesn't continue until RETURN is pressed. Whenever you use an INPUT, always ask specifically for what you want. The worst thing for an inexperienced computer user is to see a question mark, and no instructions. The following program does the same thing as the previous program, but in a more efficient way.

```
10 INPUT"ENTER A NUMBER";A
20 INPUT"ENTER A WORD";A$
```

CHAPTER THREE

```
30 PRINT A,A$
40 END
```

This is the best method for using INPUT, and it is fairly standardized. It will work with most BASIC languages (except Atari BASIC) without modification. You should use INPUT whenever you expect more than one character to be entered at a time. One drawback to this method is that it stops the execution of the program. This is not always desired. Also, sometimes you'll want to limit the time the user has to enter data. In these last two examples a GET command would be more useful.

GET

The GET command is also very popular, but it allows only one character to be entered at a time. An advantage of GET over INPUT is that the RETURN key need not be pressed. The GET command extracts the first character from the keyboard buffer. This buffer is where the ASCII value (a numbered character code) of a character goes after its key has been pressed. This buffer becomes more important when we start to talk about the PEEK statement. The GET takes only one character at a time, which can be good or bad, depending on the situation. This command will work with numeric variables, but I recommend using string variables at all times. When you are using GET and asking for numbers, but get letters, then ? REDO FROM START will appear on the screen. The GET is quite a bit more complicated than the INPUT and takes some getting used to. This program should help you understand how to use it.

```
10 PRINT"HIT ANY KEY"
20 GET A$
30 IF A$="" THEN 20
40 PRINT A$
50 END
```

This small program GETs a character from the keyboard and prints it to the screen. Line 20 is where the actual GETting is done. Line 30 prevents the program from going further if no key is pressed. If the program user does not press a key, then A\$ would equal "" – called a null string. If a key is pressed, the program goes on. The purpose of line 30 is to stop and wait for some kind of character to be entered. If you don't want to stop the program, then line 30 would be left out. This allows other action to take place while the user provides some data. Another use of the GET is to let someone

go through a program at his or her own pace. This statement is often used at the bottom of a screen of instructions. It allows you to read them, and then move on to the next screen.

The PEEK Method

The final method has not been widely documented. It involves PEEKing at either location 197 or location 201. These are addresses which return a numeric value for the key which is currently being pressed. The method is quite simple. After typing in and running this program, hit various keys and see their effect.

```
10 PRINT "{CLEAR} "
20 PRINT "{HOME} "; PEEK (197)
30 GOTO 20
```

This is an input method which is used in many games. You may wonder what these numbers are. There is a special code for these PEEKs which is different from both the ASCII code and the POKE code. Each key is assigned a certain value which is the same whether it is shifted or not. The value for all keys up is 64. Table 1 shows the code for each separate unshifted character. Using one of these two PEEKs is the most effective way to control game animation from the keyboard. It's better than the GET for these purposes because it tells you what is currently pressed, not what was pressed. This method does not extract characters from the keyboard buffer. This could cause problems if you have a GET or INPUT following one of these PEEKs. The characters pressed for the PEEK were entered into the keyboard buffer, but weren't used. When you do come to a GET or INPUT, the keyboard buffer will empty. This will cause the program to go right through a GET, or to empty unused characters at an INPUT.

Table 1. VIC Character Code for PEEK(197)

←	8	A	17
1	0	S	41
2	56	D	18
3	1	F	42
4	57	G	19
5	2	H	43
6	58	J	20
7	3	K	44
8	59	L	21
9	4	:	45
0	60	;	22

CHAPTER THREE

+	5	=	46
-	61	RETURN	15
£	6	Z	33
HOME	62	X	26
DEL	7	C	34
Q	48	V	27
W	9	B	35
E	49	N	28
R	10	M	36
T	50	,	29
Y	11	.	37
U	51	/	30
I	12	CRSR DOWN	31
O	52	CRSR RIGHT	23
P	13	f1	39
@	53	f3	47
*	14	f5	55
↑	54	f7	63
		SPACE	32

Amortize

AMIHAI GLAZER

This program demonstrates an extraordinary method of getting numbers from the user. You can enter expressions such as $3 + 2$ as well as the simple number 5.

You're planning to buy a new house. Or perhaps a new car. But money is short and you must take out a loan. What is the monthly payment on the loan? What is the total interest charge? How much interest can you deduct from your income tax in the first year? Answers to these and other questions are provided by the program "Amortize." As an added bonus, the program incorporates some techniques you may want to use in your own programs.

Key in the program. On line 63993, simultaneously press the SHIFT key and the letter "O" key; this is an abbreviation for GOTO. On line 63992 the PRINT statement consists of a quote mark, a blank space, pressing the CTRL and the "2" keys simultaneously, and finally a quote mark. On line 63996, to enter the PRINT statement type a quote mark, then press the CTRL and the "7" keys simultaneously, then press the space bar seven times, and then close with a quote.

Once the program is in memory, type RUN and you will be prompted for the input. Notice that you can type as input not only numbers, but expressions as well. For example, suppose we let the loan be for ten years, so that for the number of months we enter 10×12 ; we let the interest rate be $13 + 3/8$, and the amount of the loan (the principal) is \$50,000. Your friendly VIC-20 will respond by showing that the monthly payment on the loan (PMT) is \$757.65. You will then learn that after your third payment (MONTH = 3) you still owe \$49392.20, that you have paid a total of \$1665.15 interest over these three months, and that \$552.80 out of your third payment went to pay interest.

Look at the results for the last month, month 120. You will find that the total interest paid on the loan is \$40918.87. (Yes, that sure is a lot of interest.) Don't let the small amount of principal remaining, 87 cents, bother you; such inaccuracies are inevitable when you can't make monthly payments including a fraction of a cent. One final caveat: some consumer lenders use the Rule of 78s to determine

the reduction in principal each month. Therefore, the results the program gives you for any but the last month may be slightly different from what the bank may tell you. But the program will still give you the correct value for the monthly payment, and the correct value for the total interest charge.

The INPUT Technique

That's it for you folks who want to use the program without worrying about how it works. As mentioned, the user's input can be in the form of an expression, not merely a number (which is what the INPUT statement allows). Here's how this is done. Suppose we want to get a value for variable N. In statement 50, the computer printed out the characters "N = ". The user types in any expression, say $10*12$. Lines 63990 and 63991 accept the characters for this expression and print it out. The screen will now show $N = 10*12$. (The POKES into locations 204 and 207 allow the cursor to be shown when the GET is invoked.) We then switch (in statement 63992) to printing in white so the user will not be confused by the tricks we are about to play.

In statement 63993 the computer prints G Γ 63996. In statements 63994 and 63995 we POKE (into the keyboard buffer) instructions to go up to the screen line which says $N = 10*12$, to execute that line, to go to the screen line which says G Γ 63996, and to execute that line. These instructions are executed when the END in statement 63995 is encountered.

Having executed the instruction on the screen to GOTO 63996, the computer is now executing that statement. The computer switches back to printing in blue, erases from the screen all the garbage which it had printed in its machinations, and returns to the calling program.

The subroutine which starts in statement 63990 can be used in any program you wish. The calling sequence is exactly as shown in statement 50.

There is another useful technique in lines 132-136. These instructions allow the user to stop execution by pressing any key, and to continue execution by pressing that or any other key; the instructions transform the keyboard into a toggle switch. The logic is simple: if no key is pressed when line 132 is encountered, the program does not stop. If any key is pressed, the program waits until all keys are released, and then waits until a key is pressed. Execution then continues.

```

1 REM AMORTIZE --BY
2 REM AMIHAI GLAZER
3 REM UNIV. OF CALIF.
4 REM IRVINE,CA.92717
5 DEF FNR(X)=INT(100*      X+.5)/100
10 PRINT "{CLEAR}{REV}AMORTIZE"
20 PRINT "{03 DOWN}"
30 PRINT "NO. OF PERIODS"
40 PRINT "  (IN MONTHS)"
50 PRINT "N= ";GOSUB 63990
60 PRINT "ANNUAL %INTEREST RATE"
70 PRINT "AR=";GOSUB 63990
80 MR=AR/1200
90 PRINT "PRINCIPAL"
95 PRINT "P=";:      GOSUB 63990
100 PMT=(P*MR)/(1-(1+      MR)^(-N))
105 PMT =FNR(PMT)
110 PRINT "{02 DOWN}PMT=",      FNR(PMT)
111 PRINT "{DOWN}PRESS RETURN KEY"
112 PRINT "TO CONTINUE OR STOP"
113 GETA$:IF A$=""      THEN 113
120 PRINT "{02 DOWN}"
130 FOR I=1 TO N
132 GET A$:IF A$=""      THEN GOTO 140
134 GET A$: IF A$<>""      THEN GOTO 134
136 GET A$:IF A$=""      THEN GOTO 136
140 RDUE =FNR(P*MR)
150 CUMR=FNR(CUMR+RDUE)
160 P=P-PMT+RDUE
170 PRINT "{REV}MONTH=";I
180 PRINT " PRINCIPAL =" ;FNR(P)
190 PRINT " TOTAL INT.=" ; (CUMR)
200 PRINT " INT. DUE =" ; (RDUE)
210 NEXT I
220 END
63990 POKE 204,0:POKE 207,0:GET A$
63991 IF A$<>CHR$(13) THEN PRINT A$;:GOTO 639
90
63992 PRINT " {WHT}"
63993 PRINT "GO63996"
63994 POKE 631,145:      POKE632,145:POKE633, 14
5:POKE634,145:POKE 635,13
63995 POKE 636,145:      POKE637,145:POKE638, 13
:POKE198,8:END
63996 PRINT"{02 UP}":FOR ZZ =1TO3:PRINT"{BLU}
":NEXT:PRINT"{03 UP}"
63997 RETURN

```

Append

WAYNE KOZUN

An Append routine combines two programs together. With VIC's "relocating loader" it's easy, and this program makes it automatic.

Sooner or later you'll probably have two or more programs you want to combine into one. I often use the same subroutine in many of my programs. All the games I have made use joysticks as controllers. Rather than type in the subroutine each time, this program permits you to append the subroutine onto the main program. I've seen a number of append programs for the PET, but I've yet to see one for the VIC. I was in need of such a program so I decided to write one.

To my surprise the program was incredibly easy, although it appears complicated. This simplicity is due to the "relocater" in the VIC. This feature loads a program into the start of BASIC (addresses 43-44), wherever that may be, regardless of the saving locations written in the header of the program. Since the PET doesn't have this feature, an append is much more complex.

There is no limit to the number of programs which can be appended except, of course, that they must fit in the VIC's memory. I believe an append will work with any amount of memory, though I've tried it only on my 5K VIC. This program was written with tape in mind, but I see no reason why it shouldn't work with a disk drive. This article was written for someone with a fair amount of knowledge about the dynamic keyboard, etc., and it helps if you know a bit of machine language.

The Inner Mechanism

What makes this program tick is the dynamic keyboard feature of the VIC, when you print to the screen a program line, or a direct statement, and then force a RETURN over it. This works just as if you had typed in the line and pressed RETURN. Addresses 631-640 contain the keyboard buffer. It is here that RETURNS, having a value of 13, are POKEd. Location 198 is the number of characters in the keyboard buffer, and this must be POKEd with the number of RETURNS we put in the keyboard buffer.

When the program reaches the END statement, it returns control to the keyboard. The computer then checks to see if there is

anything in the keyboard buffer (location 198). If there is, it is spilled on the screen, where it goes over the LOAD and the POKES. Here again, the VIC has an advantage over the PET. The LOAD and the POKES are invisible to the user, giving the program a neater appearance. This invisibility is achieved by printing these direct commands in the same color as the screen background, white.

The computer recognizes only numbers. All program lines are converted into numbers and are stored this way. The end of the program is denoted by two consecutive bytes of zero, after the last program line in memory. After printing the direct commands to the screen, the computer subtracts two from the pointers to the end of program (addresses 45-46) and puts this value into the pointer to the start of BASIC (addresses 43-44). The computer then ENDS, dumping the keyboard buffer, which causes the LOAD. The next program automatically loads just on top of the previous program, thanks to locations 43-44. These two pointers are then reset to their original values when the load has been completed. This process is repeated for each of the programs which are to be appended.

Making An Append

The first thing to do is, of course, type in the program. Then proofread it. Certain errors could send your VIC into the land of evermore. When you're reasonably certain that it's correct, you should save it. After doing all this, you're finally ready to run the program. Put the tape with the main program into the recorder, and RUN. Repeat this process with the rest of the programs to be merged. When you are finished with all the programs, delete the APPEND program. The next thing to do is to check if there are any common line numbers. If so, then one of the common numbers must be changed. Otherwise, on GOTOs and GOSUBs the execution would go to the first line, and your program would not work properly. The next step would be to write a program which could renumber as well as append, perhaps using the renumbering routine elsewhere in this book.

This is not the greatest append program, but it is one of the shortest I've seen and it serves its purpose. If you have any improvements or find any bugs (a so-called APPENDicitis), then please write me.

Wayne Kozun
861 Book Rd. East
Ancaster, Ontario
Canada L9G-3L1

CHAPTER THREE

```
1 POKE 36879,27:A=PEEK(44)
2 PRINT"{WHT}{CLEAR}{03 DOWN}LOAD{10 DOWN}{0
  3 LEFT}POKE 43,1:POKE 44,";A;"{HOME}{
  BLU}":FOR I=631 TO 636:POKE I,13:NEXT I
3 POKE 198,6
4 IF PEEK(45)<2 THEN POKE 43,PEEK(45)-2+255:
  POKE 44,PEEK(46)-1:END
5 POKE 43,PEEK(45)-2:POKE 44,PEEK(46):END
6 END
```

Printing The Screen

C. D. LANE

This program will produce an exact copy of the VIC screen on a VIC 1515 printer. Using dot graphics, it can even reproduce custom characters.

This program prints whatever is on the VIC's screen onto the VIC 1515 printer. Unlike the routine supplied in the printer manual, this one gives you an exact bit by bit printing of the screen. It is very general and can be modified for other printers. Due to its generality, it is very slow, but the results are worth the wait.

The VIC Printer

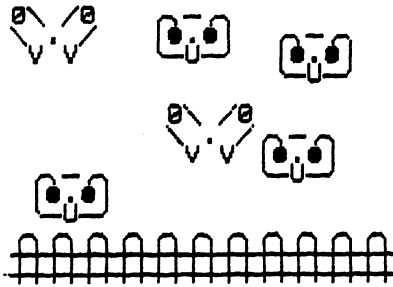
The VIC printer uses characters that consist of seven rows of five dots. In graphic mode, the printer allows the user to make characters with seven rows of six dots. If you look closely at the printer's representation of the PET/VIC graphic characters, you will see that they do not match exactly what you saw on the screen. The reason is that the screen characters consist of eight rows of eight dots, and the printer characters are scaled down to fewer dots. One obvious example is the four by four checkerboard character which comes out three by three on the printer.

Another problem with using a routine that just prints the characters by character code is that the printer puts spaces between lines that do not exist on the screen, so your drawing will be elongated and striped with white lines. The printer also has to be told which character set you wish to use, despite what is on the screen. Fortunately, the following program solves all of these problems and also allows the printing of user-defined characters.

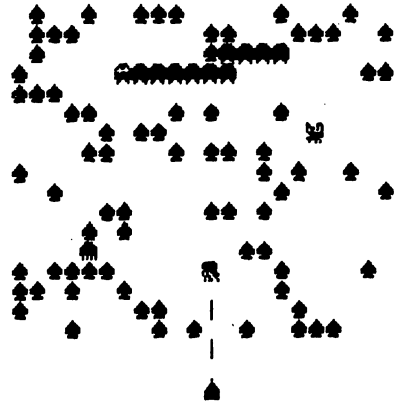
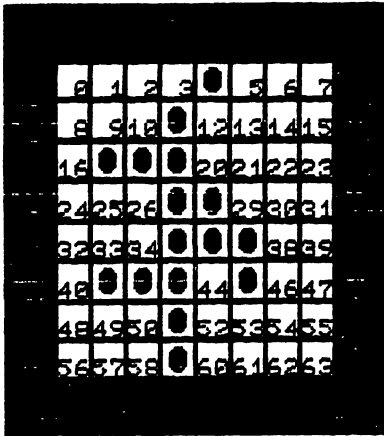
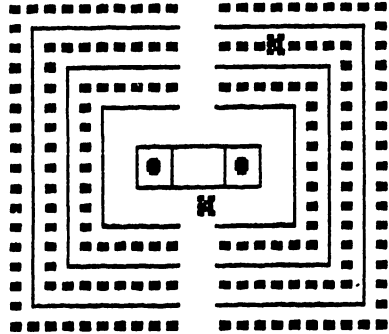
The program overlays a seven-row by six-dot mask over the eight-row by eight-dot characters, picking up where it left off each time so no bits are missed. At any given time, the character being printed may be part of one row of characters on the screen and part of another. A character printed by the printer no longer corresponds to a character on the screen, but rather to an area being examined. The program does calculations for each bit on the screen.

Before printing, the program determines where the screen is in memory. It then sets a flag if user-defined characters are being used. Next it determines where the character set resides in memory. The

CHAPTER THREE



CHANCES LEFT: 4



```

1  HIGH=23:WIDE=22:ROM=2^15:ADR=36869:OPEN4,4
   :PRINT#4,CHR$(8)
2  SIZE=((PEEK(36867)AND1)+1)*8:SC=PEEK(648)*
   256:UC=PEEK(ADR)/8AND1
3  CH=(1-UC)*ROM+(PEEK(ADR)AND7)*1024:LIM=INT
   ((5+WIDE*8)/6)*6
4  FORL=0TOHIGH*SIZE-1STEP7:FORC=0TOLIM:A=0:X
   %=C/8:BIT=2^(7-C+X%*8):FORR=LTOR+6
5  Y%=R/SIZE:CC=PEEK(SC+X%+Y%*WIDE):MEM=CH:IF
   UCTHENIFCC>127THENMEM=ROM:CC=CC-128
6  IFR<HIGH*8THENIFPEEK(CC*SIZE+MEM+R-Y%*SIZE
   )ANDBITTHENA=A+2^(R-L)
7  NEXTR:PRINT#4,CHR$(128-(C<WIDE*8)*A);:NEXT
   C:PRINT#4:NEXTL:CLOSE4:END

```

The Confusing Quote

CHARLES BRANNON

What to do with the perplexing quote character.

As you type in a program, you will eventually come to a place where information is in quotes. This tells the computer to "take this exactly as shown," instead of interpreting it. For example, the instruction `PRINT "PRINT"` causes the word `PRINT` to be displayed on the screen. The `PRINT` in quotes is entirely different from the command `PRINT`. What, however, do you do when you're typing in a line like:

```
10 INPUT "WHAT IS YOUR NAME";N$
```

and you make a mistake at the beginning of the line? You just cursor-left to the error, and correct it. Right? Nope. What you get are a bunch of reverse-field vertical lines. These are *control characters*, but that explanation doesn't help you retype that error.

When you typed that first quote you entered the twilight zone of *quote mode*, which is both one of the most frustrating and most useful features of the VIC. The trick is that cursor keys are not only for use in screen editing, but can also be programmed. When the VIC comes to one of those reverse-field vertical lines, it will attempt to actually move the cursor left one space. This can be used to produce animation. When a character is printed on the screen, the cursor moves to the right one space, just as on a typewriter. If, however, you move it back with a programmed cursor-left, you can replace the old character with a new one. Try this line:

```
10 FOR I = 1 TO 20:PRINT "·||"; :NEXT I
```

Other cursor controls can be programmed as well. The most commonly-used one is the clear-screen character. This is at the start of most programs, and it appears as a reverse-field heart. Actually, all control characters on the VIC are in reverse-field. Cursor-down (Q) can be used to skip down to any line quickly, without having to print a blank line. Hence the line:

```
10 PRINT:PRINT:PRINT:PRINT
```

can be replaced by

10 PRINT"OOOO";

Used in conjunction with the HOME character, cursor down can act like a "vertical TAB statement." At the start of your program, define CDS (or any string, really) to be equal to HOME and 21 cursor-downs. Now you can place the cursor on any line with PRINT LEFT\$(CDS,L), where L is the screen line, from one to 22. Reverse field on and off are also easy to use; just insert the appropriate characters before and after the text you want highlighted. The color control keys are used similarly, except that while reverse-field is cancelled by a carriage-return, the color command remains in effect until changed. Has your display ever disappeared? Don't despair, you probably changed the text color to white (CONTROL2), and if the background was white, everything would disappear. Just type CTRL some other color to regain it, or reset with STOP/RESTORE.

Okay, now you're using control characters to do amazing things, but you may be experiencing another problem – you can't make the VIC print them. The problem here is that you are not in quote mode. Here is exactly how quote mode works:

- 1.** If you type an odd number of quotes, you are in *quote-mode* – all cursor controls (except DELeTe) will show up for better or worse.
- 2.** An even number of quotes, two or four, or none at all, keeps you in the *edit mode*, where you can move the cursor anywhere and type.
- 3.** A special way to get into quote mode is with the INST key. When you insert a gap in text, you are temporarily in quote mode. If you type any control key, it will be printed. This is useful for placing cursor controls inside an already-typed line.

Finally, if you are going crazy trying to figure out what those quotes are doing to your line, just type SHIFTED RETURN to escape to the next line. SHIFTED RETURN does not act like an ENTER key. It just moves to the next line, and cancels reverse field and quote mode. You can then cursor up to the mangled line and fix it.

Remember, one of the VIC's strengths is its ability to manipulate the cursor, colors, and even select upper/lower or uppercase/graphics. Don't neglect this feature. And you can quote me on that.

Alternate Screens

JIM BUTTERFIELD

One of the more exciting things about the VIC is the control you can exercise over the screen, and there are many ways to do it.

The screen itself doesn't move, of course. It stays wherever you have plunked your TV set or monitor. What we are doing is changing the location from which the screen gets its information.

The Screening Process

The information that appears on the screen is taken from the memory of the VIC. Each character on the screen corresponds to a value stored in memory. Each screen location is linked to a specific memory location. Here's the trick we are after: we plan to change the links so that each screen location is fed from a different memory cell.

If we can do this successfully, we'll be able to store two completely separate "screens" in memory. We can then switch the display back and forth between these two screens so as to create special effects or a useful "dual" display.

There is a location in VIC's memory which controls where the screen display is linked to memory. We can easily switch the contents of that location, which would cause the screen to flip to the new display. But that's not enough.

Working On The Screen

We must also change the working pointers in the VIC – the ones that put new characters on the screen. It might not help us much if we switched the display to a new location but kept on typing characters into the old one.

Another small job that we will need to do: we must set aside the extra memory we need for the second screen. This will reduce our paltry 3500-odd bytes to less than 3000, but it's a necessary sacrifice. After all, we don't want BASIC messing around in this screen area and, unless we say otherwise, BASIC will use as much memory as it can find.

The First Step

Try typing `PRINT PEEK(56)` and pressing RETURN. You should see a value of 30 printed. Any other value means that your VIC

doesn't have the normal 5K of memory, and the rest of this procedure won't work.

The value of 30 tells us the address of the "page" where BASIC stops. A page is a chunk of 256 bytes. We're going to take away two more pages in order to free enough bytes for the second screen.

Type `POKE 56,28:CLR` and press RETURN. Now we've stolen away 500-odd bytes from BASIC. If you don't believe it, type `PRINT FRE(0)` and see. Don't worry too much about the loss. Everything will restore to its normal state when you turn the power off; we're just making a temporary change.

Flipping Over

You'll need to type four crowded lines. Excuse the congestion; if I put neat spacing in we couldn't get it all to fit, and then we'd end up stuck halfway between screens. Type the following as a single block without pressing RETURN:

```
POKE36866,22:POKE648,28:
FORJ=217TO228:POKEJ,156:
NEXT:FORJ=229TO240:
POKEJ,157:NEXT
```

Proofread it carefully; one mistake and you'll have to power down and do it all again. When you're sure it's right, press RETURN.

Bingo! You're into the other screen. It looks like a mess, for this screen area was never cleared. Clear the screen and type your name and perhaps one or two other messages; you might like to change the color of the characters. Now to prove that we really have two screens, we're going back home.

Type in the following, again as a single block:

```
POKE36866,150:POKE648,30:
FORJ=217TO228:POKEJ,158:
NEXT:FORJ=229TO250:
POKEJ,159:NEXT
```

Proofread, press RETURN, and you're back in the first screen.

Picky Stuff

The first annoyance that you'll notice is that, right after you switch screens, the VIC prints READY, probably right in the middle of the new screen. This isn't much of a problem; when you write the above statements as a program they will execute without the READY being printed. Another advantage of writing this into a program is

that you don't have to crowd everything into a single line.

The second problem is less visible to begin with, but more serious. All the linked lines have been broken apart; instead of one four-row line with our massive POKE sequence on it, we'll have four individual lines. This may not matter on some kinds of display, and might even be nice if we wanted that effect. But we often want to keep the screen exactly as it was when we left it.

The trick is in locations 217 to 240: to restore the screen, we must restore these values exactly as they were. That will take a little extra coding.

A Program

Here's a little program to do all the above jobs. We'll use the VIC's F1 key to switch between screens.

```

100 REMDUAL SCREEN JIM BUTTERFIELD
110 POKE56,28:CLR
120 DIML%(23)
130 GOSUB400:PRINTCHR$(147):GOSUB400
140 Z$=CHR$(133)
200 GETX$:IFX$=Z$THENGOSUB400
210 PRINTX$;:GOTO200
400 REM SWITCH
410 S=PEEK(648)
420 IFS=28THENS=30:T=150:GOTO500
430 IFS=30THENS=28:T=22:GOTO500
440 STOP:REMARK:ERROR
500 POKE648,S:POKE36866,T
510 FORJ=0TO23
520 V=PEEK(J+217):POKEJ+217,L%(J):
530 L%(J)=V
540 NEXTJ
550 PRINT:RETURN

```

Further Thought

It works. Can you have three, four, or five screens? There seems to be no reason why not. Unfortunately, you'd need to compromise to make it work; there are only two screen color tables. We haven't even talked about these – they took care of themselves in our example.

If we omitted the PRINT statement from line 550, we'd find another oddity: the VIC doesn't really switch screens until a line is complete. It doesn't look for a new location to put characters until

CHAPTER THREE

RETURN or some other event signals the end of a line. We've taken the simple way out here; more elaborate coding would be needed to reinstate the screen in mid-line.

Let's leave this exercise at its present stage of development. It's useful as it stands; it doesn't gobble up too much memory; and it's fairly simple.

For those who need to know mechanisms: 36866 is the location that actually switches the display. 648 tells the VIC where to find the whole screen. The values at 217 to 240 do two jobs. First, they tell the VIC roughly where the screen line rows have been linked together to form a long line. I call this series of values "the screen wrap table," but the name is less important than an understanding of what it does. The color control locations are at a high location – above 30000 – and, happily, we don't need to deal with them here.

Timekeeping

KEITH SCHLEIFFER

How to use VIC's timekeeping functions, TI and TIS, to simulate a clock or to create controlled delays during a program RUN.

The friendly computer guide that comes in the box with your VIC-20 mentions several interesting features that the casual reader can easily miss. In my most recent rereading, I discovered the timekeeping feature of the VIC. The computer can keep real clock time, count elapsed time, or time controlled pauses during program execution.

The clock is available as the reserved variables TI and TIS. TI actually counts time passing. TIS is a string variable, which depicts this time count in HHMMSS format (hours, minutes, and seconds, without any punctuation) on a 24-hour clock.

How does the VIC do this timekeeping? When the computer is first turned on, the timekeeper initializes at 000000 (midnight). You can then set it to act as a clock by assigning to TIS a string representing the correct time. For instance, if I initialize the timekeeper as a clock at 1:29:30 in the afternoon, I would enter the statement:

TIS = "132930"

The VIC would convert this to 48570 seconds after midnight, multiply by 60 and assign:

TI = 2914200

and continue counting from there. TI is counted in one-sixtieth second intervals; that is, when TI has increased by 60, one second has passed. The time count is kept in memory locations 160, 161, and 162.

Once you have set the correct time, you can check it whenever you wish by entering:

PRINT TIS

and the VIC will display the time, again in HHMMSS format. I like to set TIS to keep clock time, and check it occasionally, so my wife doesn't have to complain about getting less attention than the computer. The timekeeper can be used in programming to control operations at scheduled times during the day, such as periodic data-

CHAPTER THREE

collection from an experiment, or to control your lights in a household security program.

To use the VIC to count elapsed time, you cannot start and stop the time counter. To get around this problem, you must run a second variable to count time in parallel with TI, then stop counting with that second variable when the timed period is over. The following program uses the "hit any key" concept to start and stop timing:

```
100 GETA$:IFA$=" "THEN100
110 TS=TI
120 PRINT"TIMING"
130 TC=TI:GETA$:IFA$=" "THEN130
140 TE=(INT((TC-TS)/6+0.5))/10
150 GOSUB400:PRINTT$
160 END
400 REM CONVERTS SECONDS TO HH:MM:SS.S FORMAT
410 HI=INT(((TE/60/60/24)-(INT(TE/60/60/24)))*
24)
420 B1=STR$(H1)
430 H$=MID$(B$,2,2):IFH1<10THENH$="0"+MID$(B$,
2,1)
440 T3=TE-(H1*60*24)
450 M1=INT(((T3/60/60)-(INT(T3/60/60)))*60)
460 B$=STR$(M1)
470 M$=MID$(B$,2,2):IFM1<10THENM$="0"+MID$(B$,
2,1)
480 T2=T3-(M1*60)
490 S1=INT(((T2/60)-(INT(T2/60)))*60)
500 B$=STR$(S1)
510 S$=MID$(B$,2,4):IFS1<10THENS$="0"+MID$(B$,
2,3)
520 T$=H$+": "+M$+": "+S$:RETURN
```

Line 140 converts TE to the elapsed time in seconds and rounds off to the nearest tenth. The subroutine starting at line 400 will convert this to "clock" display, complete with colons in HH:MM:SS.S format, down to tenths of seconds. A simpler approach would use TIS by assigning to it the elapsed time value and immediately printing it:

```
140 TE=TC-TS
150 TI=TE:PRINTTI$:END
```

You won't want to use this method if you are using TI as a real clock, or if you're relying on the timekeeper to track more than one period at once.

You can use the timekeeper for the scoreboard in a game, either by displaying stopwatch time or TIS, to show time passing, or by calculating time remaining and displaying a countdown timer.

The following program is a version of the countdown timer.

```

100 PL=5:REM PERIOD LENGTH 5 MINUTES
110 PS=TI:REM PERIOD STARTS NOW
120 PF=PS+PL*60:REM PERIOD FINISH TIME
130 TR=PF-TI:REM TIME REMAINING
140 GOSUB400
150 PRINT"{CLEAR}"T$
160 IFTI<PFTHEN130
170 END
400 REM CONVERTS SECONDS TO MM:SS FORMAT
440 T3=INT(TR/60+.5)
450 M1=INT(((T3/60/60)-(INT(T3/60/60)))*60)
460 B$=STR$(M1)
470 M$=MID$(B$,2,2):IFM1<10THENM$="0"+MID$(B$,
    2,1)
480 T2=T3-(M1*60)
490 S1=INT(((T2/60)-(INT(T2/60)))*60)
500 B$=STR$(S1)
510 S$=MID$(B$,2,2):IFS1<10THENS$="0"+MID$(B$,
    2,1)
520 T$=M$+"":S$+":RETURN

```

The most valuable feature of the timekeeper is the ability to control the length of pauses made during execution, independent of the program lines being executed. The friendly computer guide shows how to make delays by using a FOR ... NEXT loop with the statements:

```
FOR I=1 TO 100:NEXT I
```

The major problem with this method is that it ties up the whole program while you pause. You can insert program lines for execution during the loop, but then some guesswork and experimenting will be necessary every time you program to obtain the desired pause. Frequently you will have to compromise between the statements you want to execute and the time you can allot to the pause. Finally, if the lines executed during the pause contain the decisions with varying amounts of program to be executed based on the decision, the length of the pause becomes unpredictable.

Getting Control Over Pause

The timekeeper counts independently, on a steady basis, and allows you to assume control of the length of a pause, while permitting other parts of the program to continue. To do this you simply note the time the pause begins and add the desired pause length, giving the time the pause will end. An IF decision watches for the clock to exceed that end time, and you can run other parts of the program

CHAPTER THREE

while the pause is in progress. The decision watching for the end of the pause must be made with a reasonable frequency, so the number of statements you can execute between repetitions of the end-time decision will depend on how long the pause is and how exact you want the measurement of the pause to be.

As a very conservative rule-of-thumb, allow 20 80-character (multiple statement) program lines to reach the end-time decision at an interval of about ten percent of the total pause length. For example, if I pause for about ten seconds, I can allow up to one second, or about 20 program lines. Similarly, a two-second pause will allow up to four program lines between repetitions of the end-time decision. You can use a greater number of lines if they do not contain several statements each.

These time estimates are very rough: do some experimenting yourself to find how many statements you can squeeze in and still get accurate control of the pause length. Once you have established some rules for yourself, they should be useful in all your programming.

As an example of the pause, let's say that I'm writing a game program in which we explore a dungeon. If someone casts a magic spell of darkness, then I want to give no visual clues for the length of the spell – say 20 seconds – while the action of the program continues. The following segment of a program will provide that effect:

```
100 DEF FN PS(T2)=TI+(T2*60)
350 REM THE SPELL IS CAST
360 GOSUB 900:P1=FN PS(20)
370 REM P1=TIME TO END BLACKOUT
380 REM THE
390 REM PROGRAM
400 REM CONTINUES
410 REM RUNNING
420 REM WITH A
430 REM BLACK
440 REM SCREEN
490 REM (UP TO FORTY PROGRAM LINES)
500 IF TI>P1 THEN GOSUB 902:GOTO 800
780 GOTO 380
800 END
900 POKE 36879,8:FOR I=38400 TO 38906
901 POKE I,0:NEXT I
905 RETURN:REM BLACKOUT MAKER
920 POKE 36879,78:RETURN:REM BLACKOUT LIFTER
```

This application uses the function PS to relate the desired pause length (T2) to a future time value (P1), which defines the end of the blackout.

Another application of the pause timer can limit how often I may perform an action. I'm writing a game in which the player fires a laser cannon that takes five seconds to recharge before it can be fired again. The line which times the firing interval looks like this:

```
350 IF PEEK(197)=35 AND TI>P1 THEN GOSUB 800:P1=TI+(5*60)
800 RETURN:REM VISUAL AND SOUND EFFECT FOR LASER FIRING
```

Here there is no need to worry about running the end-time decision within a set interval – the next time I want to fire the cannon, the logical AND in the decision checks to see if it has recharged. This pause method can also be used in an education program, to limit how soon the student may answer after a question appears, or may try a second time after an incorrect first answer has been entered.

If you're interested in converting existing programs to timekeeper pauses, the statement:

```
FOR I=1 TO 100:NEXT I
```

is worth about eight counts on the timekeeper, or 0.13 seconds. There will be some difference between this statement and a longer loop. For instance, modifying the statement to:

```
FOR I=1 TO 1000:NEXT I
```

This is worth 72 counts, or 1.2 seconds, not the 80 counts one might expect. This is because of the "overhead" time needed to establish the loop during execution. There may even be differences between machines. You can check your own timing with this simple program:

```
10 BT=TI
20 FOR J=1 TO 1000:NEXT J
30 FT=T1
40 ET=FT-BT
50 PRINT ET,ET/60
```

This displays the time passed in both counts and seconds. Try varying the length of the loop in line 20 to get a general idea of what the "overhead" time is on your computer.

You need to do nothing to the timer to use it as a basis for pauses. However, if you have the VIC on for long periods, or if you set TIS to keep clock time and run the program near midnight, be careful: if the pause starts before midnight and ends after, you may never reach the end of the pause, since the clock resets to 000000 at midnight. You can put in additional statements to watch for this problem and compensate for it; you can have the program reset the clock to 000000 before

CHAPTER THREE

timing any pauses; or you can ignore the possibility and hope for the best. The third option, technically unsound as it is, requires the least effort and presents no great threat.

These pause techniques have two important features: controllable pause lengths and the ability to run other, unrelated parts of the program while the pause is in effect. When you develop a program, you can select a length of pause that will not change as you add, change, remove, or relocate program statements. The pause can also be lengthened or shortened to suit your needs, without major changes in the program itself. You have made your pause independent of the program that contains it. At the same time, you can execute lines of an unrelated portion of the program while the pause is in progress, making the program independent of the pause it executes. The timekeeper in the VIC gives the programmer much better control of realism in his game and simulation programs.

Renumber BASIC Lines The Easy Way

CHARLES H. GOULD

This simple Renumber program can save you much time and grief. Note, however, that you must manually change line number references such as GOTO or GOSUB.

Until we have a programmer's aid ROM available, such as the Toolkit, here is a simple way to renumber lines in that crowded program you are writing. Simply type in the lines given below, and: RUN 10000. Better still, put this program on a separate tape, load it whenever you start a new program development; when completed, erase these lines. The program uses only 198 bytes, so 95% of your RAM is still there.

As shown, the renumbered lines start at line 10, and increment each line number by ten. *But* it does not change GOTO or GOSUB line numbers. You must manually change these after renumbering, and before running. An easy way to keep track of the GOTO and GOSUB terminal addresses is to insert "REM line #" at the end of each such statement, renumber, correct GOTO and GOSUB references to the new line numbers, and erase the REMs.

Line 9990 simply protects your program from entering the renumber routine until you want it to. In line 10010, Y7 is the starting line number, and in line 10050, the $Y7 = Y7 + 10$ sets the increment. Either can be changed. Line 10020 tests to see if we have renumbered up to, but not including, line 9990. Line 10030 changes the line number. Line 10040 searches for the next line. Y6 is the normal start of BASIC text (-1). The BASIC text lines are stored in RAM memory in this way: the first and second (low and high) bytes are a link to the next line; the third and fourth bytes are the line number (see line 10030 below); the BASIC statement is then given using tokens; the last byte of the line is a null (ASCII 0). After the last BASIC line, two more nulls are inserted to indicate end of program.

So, until we get support utility ROMs, let's make do.

CHAPTER THREE

```
9990 END
10000 REM RENUMBER
10010 Y6=4096:Y7=10
10020 IFPEEK(Y6+3)=6ANDPEEK(Y6+4)=39THENEND
10030 Y8=INT(Y7/256):Y9=Y7-256*Y8:POKEY6+3,Y9:PO
      KEY6+4,Y8
10040 IFPEEK(Y6+5)<>0THENY6=Y6+1:GOTO10040
10050 Y7=Y7+10:Y6=Y6+5:GOTO10020
```

Automatic Line Numbers

JIM WILCOX

This program is in the tradition of the famous "dynamic keyboard" technique of the PET. It can be used to make programs self-modifying. Essentially it forces the machine to print something on screen and then RETURN over it – thereby entering a line or, in direct mode (no line number), causing an action. It can be modified to delete a set of program lines automatically.

This program will allow a programmer to have automatic line numbers at any starting line number and with any increment. The screen will clear and will print "STARTING LINE #?". "A" will take on this value. The computer will then ask for the "INCREMENT?" and "B" will equal this value. The increment value will be POKEd into memory address two. The screen will clear again. The line number will be POKEd in zero and one. The computer will then print the line number.

At line number 63996, the computer will print all the characters typed in and will stay at that line number until the "RETURN" key is pressed.

The PRINT "G␣63998" is shorthand for GOTO63998. The ␣ is the shift of an "O". The interrupt driven keyboard buffer starts at 631, and I put in four cursor ups, then two RETURNS. Location 198 is the number of keys pressed. The program will then END. The computer will print READY along with the line number and statement and the "G␣63998." Once it prints READY, the characters in the keyboard buffer will be honored. The four cursor ups will move the cursor up to the line number and statement. It will then execute a RETURN which will put the line number and statement into the program. To continue the program, the cursor RETURNS over the "G␣63998" and the program will GOTO63998.

At 63998, the computer will move the cursor up two lines and will print seven spaces to blank out the "G␣63998" and "READY". The computer will then print up three lines and be just below the last statement.

CHAPTER THREE

Since the program was changed, the line number was cleared out of memory, so line 63999 sets the line number with the PEEKs and also adds the increment with the "PEEK(2)". The computer will go back to line 63995 where the new line number will be stored, and the new line number will be printed, ready for the programmer to type in the statement.

I wrote this program to use the least amount of memory and to have line numbers out of the way of any program. Do not try to alter this program until after you've typed it in exactly as shown, to prevent errors.

```
63994 INPUT "{CLEAR}STARTING LINE #";A:INPUT
      "INCREMENT";B:POKE2,B:PRINT "{CLE
      AR}";
63995 B=A/256:POKE0,(B-INT(B))*256:POKE1,B:
      PRINTA;
63996 GETA$:PRINTA$,:IFA$<>CHR$(13)THEN6399
      6
63997 PRINT"GO63998":FORA=631TO634:POKEA,14
      5:NEXT:POKEA,13:POKE636,13:POKE1
      98,6:END
63998 PRINT"{02 UP}":FORA=1TO3:PRINT"      ~
      ":NEXT:PRINT"{03 UP}";
63999 A=PEEK(0)+256*PEEK(1)+PEEK(2):GOTO639
      95
```

Putting The Squeeze On Your VIC-20: Getting The Most Out Of 5000 Bytes

STANLEY M. BERLIN

There are times when you will write a program that is so large it will need every last memory cell in your computer. This article shows several ways to reduce the size of your programs.

Five thousand of almost anything seems like a lot; a Christmas bonus of \$5,000 would make anyone happy; 5,000 jelly beans would be more than even our President could eat; 5,000 days is over 14 years. However, there are other circumstances when a quantity of 5,000 really is not so much. Five thousand raindrops would probably go unnoticed. The time when 5,000 is a really small quantity is when you are writing a program and have only 5000 bytes (memory locations) in which to do the job!

I remember working 20 years ago with a 4000-character IBM 1401 computer and feeling confident that if I had only another 25 memory locations I would be able to complete the program. Times have changed during the last 20 years, and there are many programmers who now work with *virtual memory systems* [where the computer can use disk memory as if it were RAM – Ed.] where there is no such problem as being constrained by the amount of memory. With technology moving as fast as it is, persons working on small microcomputers probably will not have to wait 20 years for a virtual memory-like system. When that time arrives, people will not have to write articles like this one. However, today, if you are writing programs for the Commodore VIC-20, you will have to live with the constraint of having only 5000 bytes worth of memory in which to do your work.

Anyone who has done any serious programming for the VIC-20 knows that it does not take many BASIC statements before you get

CHAPTER THREE

that dreaded "OUT OF MEMORY" message. Of course, the VIC-20 is nice enough to let you know when you turn it on that by the time it gets through allocating 506 bytes for the video mapping, and another 506 bytes for color mapping on the screen, and reserves memory space for such other things as tape cassette buffers, you have only 3583 bytes of memory in which to store your program.

So, there you are busily entering your new BASIC program and VIC sends you the "OUT OF MEMORY" message. What are your options? You can resign yourself to the fact that, no matter what you do, the program will never fit into memory and abandon your project. Surely, no programmer worth his or her salt would exercise this option! Another option is to run out and purchase a memory expansion unit. This is not too bad a solution except, at the time this is being written, there is no such item available. Even if there were, it would surely be a costly solution. The last option is to roll up your sleeves and dig into your program with a finely honed scalpel to perform surgery on it. That is certainly the most challenging option, and it is the purpose of this article to pass on a few points to help you in your efforts.

The items discussed will be from a BASIC programmer's point of view. Technical system information will be avoided except when it is necessary for a clear understanding of the issue. Although these suggestions are aimed at the VIC-20 owner and the VIC-20 is the computer used to validate the data, much of the information is pertinent to other computers using Microsoft BASIC.

One last point before moving to the meat of this article: many of the suggestions presented here are a tradeoff between good program documentation and the amount of memory used. Remark (REM) statements liberally scattered throughout a program provide the roadmap when you are trying to debug a program.

At this time, most VIC-20 owners probably do not have the benefit of a printer, so there is no printed copy of the program on which to make comments regarding the various routines used in a program. If you are going to do any serious programming, a printer makes the task much easier because you can follow the logic and flow of a program from beginning to end without having to enter multiple LIST statements. A disk drive provides a lot of speed and flexibility when you are using a program, but a printer is worth its weight in gold when you are trying to debug a program. If it is necessary to remove REMark statements from a program in order to conserve memory space, it is worthwhile keeping some handwritten notes concerning the program. At a minimum, you should write

down the BASIC line numbers of subroutines and major sections of the program so that you will at least have an idea of what area of the program to LIST when you want to look at or change an area of code.

REMs And Blanks

That brings us back to being confronted with the "OUT OF MEMORY" message and the first technique for buying back a few bytes of storage. REMark statements, as important as they are, require memory. They do not provide any function in your program. A REMark statement on a line by itself will require a minimum of six bytes, even if there is no text associated with the REMark. If the REMark contains textual information (as it usually does), add the length of the text to that six bytes.

The quickest way to obtain memory is simply to remove the REMark statements. Remember to write down a notation about the REMark so that the information is at least externally preserved for documentation purposes. One word of caution: if your program contains GOTO or GOSUB statements whose object is a line number containing a REMark that you removed, you will receive an "UNDEFINED STATEMENT" error message. Should you get an "UNDEFINED STATEMENT" error message, you will have to figure out where the REMark was removed and change the GOTO line number to be the line following the original REMark. This can be a long and tedious task if your program contains many statements that GOTO a REMark statement. Try to avoid this situation when you are originally writing your program by not coding GOTO or GOSUB statements which land on a line number containing a REMark.

Another way to buy back a few bytes of memory at the expense of good "internal" documentation is to remove all unnecessary blanks from the program. This makes the program a bit harder to read, but every blank removed is a byte of usable memory. That does not seem to be much, but if you add up the number of unnecessary blank spaces in your program, you will be surprised; besides, no one ever said this was going to be easy!

The VIC-20 makes it easy to remove the blanks with the use of the INST/DEL key, but remember not to remove any blanks from within quotes. Data between quote marks are called *strings* and, if you are displaying them on a television or printer, you undoubtedly need those blanks. For example, if you are displaying the message "PRESS X TO EXIT" you would not want that information displayed as "PRESSXTOEXIT".

CHAPTER THREE

The following routine was entered on the VIC-20:

```
0 NEW
100 PRINT "{CLEAR}"
200 A=2:B=3:C=4:D=5
300 IF A=B AND C=D THEN D=D+1:GOTO 400
400 PRINT "FREE=";FRE(X)
```

The results of running this program showed that there were 3465 bytes available; the program occupied 118 bytes (3583-3465 = 118). By simply removing the 15 blanks in statement 300 it became less readable:

```
300 IFA=BANDC=DTHEN D=D+1:GOTO400
```

but the results of that run showed 3480 bytes of free space, a 100% return for each blank removed. Finally, you might have observed that I did not remove the blank that separates the line number from the statement. It does not really occupy any memory and is there only for readability; in fact, if you remove that blank, you will find that BASIC will reinsert it when you LIST the statement.

Multiple Statements And Short Variable Names

Closely allied with removing blanks and removing REMark statements is putting multiple statements on a line. Your VIC-20 can display only 22 characters on a line, but BASIC will actually accept up to 80 characters. It is possible to have your BASIC statements occupy about three and one-half display lines. It is also possible to combine statements on one BASIC line by using the colon separator character. Every line number in your program contains an overhead of five bytes (for technicians: that five bytes consists of two bytes for the line number, two bytes for an internal pointer, and one byte for a delimiter at the end of each statement). You can save four of these five bytes for every statement combined on a line (the colon separator will use one of the five bytes eliminated). For example, instead of coding:

```
100 A=A+1
200 IF A>25 THEN Z%=0
```

you can save four out of the five byte overhead of the second line by coding:

```
100 A=A+1:IF A>25 THEN Z%=0
```

However, no suggestions are free of charge, and there is also

something to watch out for in this instance. You may freely combine statements for up to 80 characters, but it is possible that one of the statements you are combining might be the object of a GOTO or GOSUB statement, in which case you will receive the "UNDEFINED STATEMENT" error message. In the example immediately above, if there were another statement in the program which was "GOTO 200", line number 200 would not be in the program after combining the two lines and you would get the error message. If you had a statement that was "GOTO 100", that would not cause any problems.

Another item to watch for when combining statements is not to combine a line with a preceding line that contains an IF statement. The statement shown above is all right. However, if the two lines were reversed:

```
100 IF A>25 THEN Z%=0
200 A=A+1
```

you would not be able to combine the lines as:

```
100 IF A>25 THEN Z%=0:A=A+1
```

without altering the meaning of the statement. In this instance, the addition statement would be executed only if A were greater than 25, which is not the intent of the original.

The last item that returns a little usable memory at the expense of readability and documentation is the use of short variable names. Although variable names may be up to 255 characters, BASIC uses only the first two characters (plus the \$ and % suffixes for string and integer variables respectively). Each character in the variable name occupies a byte wherever it is used; therefore, you should limit variable names to two characters, and one character would be even more thrifty from a memory-use point of view. Limiting the names to two characters could have a side benefit inasmuch as it may eliminate a potential source of programming error. If you had two variables, one named "TAPE" and the other named "TASTE," BASIC would recognize only "TA" as the name and would, in effect, be dealing with a single variable.

Avoiding a technical discussion as to why it is so, it is usually more economical to use constants instead of variables whenever possible. A constant consists of data stored in the BASIC statement itself. Constants may require less memory than variables, especially in cases where the constant is a relatively short string. As the length

CHAPTER THREE

of the string increases, the amount of savings diminishes because the repetition of the constant also occupies memory.

Each of the following lines contains a constant:

```
100 A=A+1
    ("1" is the constant)
100 S$=D$+"SUFFIX"
    ("SUFFIX" is the constant)
100 PRINT"TOTAL=";X
    ("TOTAL=" is the constant)
```

To illustrate the savings that can be gained:

```
100 T$="THIS IS A TEST"
200 PRINT T$
300 PRINT T$
```

occupies 72 bytes of memory, whereas

```
100 PRINT"THIS IS A TEST"
200 PRINT"THIS IS A TEST"
```

occupies only 69 bytes of memory. That is not much of a savings because the string "THIS IS A TEST" is relatively long; if it were shorter, the savings would be more dramatic. The reason for this is that when data is assigned to a variable, it requires two areas of memory, but a constant requires only one (in the instruction itself).

BASIC is sometimes very shrewd as far as memory management is concerned. BASIC is smart enough to know when you have used a string and BASIC will reuse it rather than recreate it again in memory. Thus, if you have coded the statement: PRINT "THIS IS A TEST" and elsewhere in your program you coded AS="THIS IS A TEST", although memory would be required to contain pointers for the variable "AS", the actual text string "THIS IS A TEST" would not be recreated in memory (except in the instruction itself). The original text string would be pointed to by the variable. This is starting to border on the kind of technical information that this article has tried to avoid, but is interesting enough to pass on.

Don't Avoid Integer Variables

It seems that most people writing BASIC programs never bother with integer variables, yet that is where a significant savings in memory can be obtained. This is particularly true if the program contains arrays. Consider that, for each element in an array, the number of bytes occupied is as follows:

A string array = three bytes plus the length of the string per element.

A floating point array = five bytes per element.

An integer array = two bytes per element.

The contents of many arrays do not require the use of decimal points, but it is easier to code "DIM A(15)" rather than "DIM A%(15)." By using the integer form, you would save 45 bytes of memory. Suppose you were writing a program to deal a deck of cards and you defined an array to keep track of which cards have already been dealt. That 52 element array could be a string array, a floating point array, or an integer array. Obviously, there is no need for decimals in this example, so the obvious choice would be to use integers. The following program was run three times on the VIC-20, each time changing the type of array (making the variable type in statement 400 correspond to the array).

```

100 PRINT "{CLEAR}"
200 DIM X(100)           <=== First run
    DIM X%(100)         <=== Second run
    DIM X$(100)         <=== Third run
300 FOR Z=0 TO 99
400 X(Z)=Z              <=== First run
    X%(Z)=Z             <=== Second run
    X$(Z)=CHR$(Z)       <=== Third run
500 NEXT Z
510 PRINT FRE(X)

```

The differences in memory use during these three runs are very dramatic: the first run used a floating point array and occupied 601 bytes; the second run used the string array and occupied 403 bytes; and the final run, which used an integer array, occupied only 300 bytes. Is it worth the cost of 300 bytes in order to save typing in a "%" each time the variable is used?

Each of the memory conserving measures outlined so far would be relatively easy to implement using the editing capabilities of the VIC-20 once a program is written and resident in memory. The last few suggestions are harder to implement, and the savings are more indefinite.

If your program contains groupings of instructions that are repeated several times, it would be more memory efficient (and better programming practice) to incorporate those instructions once as a subroutine and GOSUB to them. If such statements are readily identifiable, you can implement this fairly easily with the following

three steps:

1. Add a RETURN statement after the first group of statements.
2. Place a GOSUB statement whose object line number is the first line number in the group immediately preceding the group.
3. Add a GOTO statement immediately after the inserted GOSUB statement whose object line number is the statement immediately following the RETURN statement.

Naturally you would delete all other occurrences of the same group of statements and replace them with a GOSUB to the newly created subroutine. This sounds very complicated, but actually is quite easy to implement and is illustrated in the fictitious routine that follows. Assume that the statements starting with line number 650 and ending with line number 710 are repeated several times in the program.

```
640 PRINT "ABC"
650 A=A+1
660 IF A=9 THEN 700
670 PRINT"MESSAGE ONE"
680 MC=MC/A
690 GOTO _____
700 PRINT "MESSAGE TWO"
710 MC=MC*A
720 IF Q+1=10
```

You can convert this to a subroutine using the technique described by adding lines 645 and 715 in the following illustration:

```
640 PRINT "ABC"
645 GOSUB 650:GOTO 720  ⚡ = = = = New statement
650 A=A+1
660 IF A=9 THEN 700
670 PRINT"MESSAGE ONE"
680 MC=MC/A
690 GOTO _____
700 PRINT "MESSAGE TWO"
710 MC=MC*A
715 RETURN  ⚡ = = = = New statement
720 IF Q+1=10
```

Another almost obvious way to decrease the amount of storage your program uses is simply to reduce the size of messages that you display on the television. For example, if your program displays the

word "TOTAL", change it to "TOT". If your program contains cards, instead of spelling out "KING," "QUEEN," and JACK," simply use a K," Q," and "J," respectively.

Another item to investigate is the use of more economical instructions to achieve the same results. Shown below is an example of how you can replace multiple IF statements with a single ON statement. The program does the exact same thing either way you write it, but using the ON statement yields a savings of 62 bytes.

The following statements

```
300 IF A=1 THEN 510
310 IF A=2 THEN 550
320 IF A=3 THEN 600
330 IF A=4 THEN 650
340 IF A=5 THEN 700
350 IF A=6 THEN 750
```

could be replaced with the single statement:

```
300 ON A GOTO 510,550,600,650,700,750
```

Consider using FOR/NEXT loops wherever possible instead of repeating instructions. Suppose you wanted to print a vertical line down the center of the screen. You could program it as:

```
300 PRINTTAB(10);"1"
310 PRINTTAB(10);"1"
320 PRINTTAB(10);"1"
330 PRINTTAB(10);"1"
340 PRINTTAB(10);"1"
350 PRINTTAB(10);"1"
360 PRINTTAB(10);"1"
370 PRINTTAB(10);"1"
380 PRINTTAB(10);"1"
390 PRINTTAB(10);"1"
400 PRINTTAB(10);"1"
```

or alternatively you could code:

```
300 FOR X=1 TO 11
310 PRINTTAB(10);"1"
320 NEXT X
```

Again, the same results are achieved, but the FOR/NEXT loop yields a savings of 118 bytes!

Overlaying

You can sometimes conserve memory by *overlays*. If your program runs in two separate and distinct phases (that is, one portion of your program completes all its work and then is never executed again), it should be possible to split your program into two sections. Have the last statement in the first section issue the statement "LOAD PHASEII" (assuming your second phase is named "PHASE II"). When the first section completes its job, the last instruction would load the next phase for execution.

This assumes that you have written PHASEII onto the cassette tape immediately after PHASEI. This is called *overlaying*; it lends itself well to a disk-oriented system, but there is no reason not to use it with tape also. You should be aware that any variables used in the first phase will not be available in the second phase. (However, even though this should work, the author has not yet been able to do it successfully.)

If you still need memory, you may be able to put some of the data used in your program on a cassette tape and read the data in during program execution. This technique will involve your writing a special program to create the data tape, but in some instances, this can yield substantial memory savings. "Custom Characters for the VIC" (reprinted in this book) contains a program which lends itself very well to illustrating this memory-saving technique.

This program contains a number of DATA statements and is shown below:

```
170 READ X:IF X>0 THEN 190
180 FOR X=X TO X+7
182 READ J
184 POKE I,J
186 NEXT
190 GOTO 170
340 DATA 7168,24,24,36,60,102,66,66,0
350 DATA 7176,124,34,34,60,34,34,124,0
360 DATA 7184,126,34,34,32,32,32,112,0
600 DATA 7376,0,0,0,0,0,0,0,0
610 DATA -1
```

You would have to write a program to write the data to cassette tape (and ideally you would write the data immediately after the program you saved that will be using that data). The original program can be easily modified to create the data tape, and an example could be:

```

100 OPEN1,1,2,"DATATAPE"
170 READ X:PRINT#1,X:IF X>0 THEN 190
180 FOR X=X TO X+7
182 READ J:PRINT#1,J
186 NEXT
190 CLOSE1
200 PRINT"DATA TAPE CREATED"
210 END
340 DATA 7168,24,24,36,60,102,66,66,0
350 DATA 7176,124,34,34,60,34,34,124,0
360 DATA 7184,126,34,34,32,32,32,112,0
600 DATA 7376,0,0,0,0,0,0,0,0
610 DATA -1

```

You would then substitute INPUT#1 statements for READ statements in the original program, but the results would be the same. If you wanted to use the concept in Mr. Malmberg's article in your own program, but needed additional memory, this technique would provide you with a significant amount of memory.

The modified program would be:

```

165 OPEN 1,1,0,"DATATAPE"
170 INPUT#1,X:IF X>0 THEN 190
180 FOR X=X TO X+7
182 INPUT#1,J
184 POKE I,J
186 NEXT
190 GOTO 170
340 REM-NO DATA STATEMENTS-RESULTS IN
350 REM-A SIGNIFICANT REDUCTION IN MEMORY

```

Let's assume that you have exercised all the memory-saving procedures outlined and your program still requires additional memory. The last thing to do is carefully investigate the logic of your program. Are there statements that are never executed (perhaps left over from an initial idea that was abandoned)? Naturally, you should remove them. Are there better ways to implement a procedure that might reduce the number of instructions necessary to accomplish some objective? Sometimes being able to buy back just five or ten bytes of memory will allow your program to run.

One final point: there are instances when you have worked for hours on a large, complicated program and you decide to save it on a cassette tape. You type in the command "SAVE MYPROGRAM",

CHAPTER THREE

press ENTER, and lo and behold, instead of receiving the message to press the play and record buttons, you instead get the message "OUT OF MEMORY." This is one of the most frustrating events possible when writing a program because you do not want to lose the hours of effort already expended. Try issuing the "SAVE" command with a shorter filename; this usually works. Instead of entering "SAVE MYPROGRAM", use "SAVE A".

It takes some effort, but using the techniques outlined here could mean the difference between being able to do what you want with your VIC-20 or being continuously confronted with an out of memory condition. With a little patience you might be able to find that needle in the haystack. If all else fails, don't give up hope. Remember that your VIC-20 has the capability of being expanded to 32K!

An Easy Way To Relocate VIC Programs On Other Commodore Computers

GREG and ROSS SHERWOOD

This is an "automated" VIC to PET loader. It will move a VIC program LOADED into the PET to the proper place. Just type SYS 926.

BASIC programs generated on the VIC-20 start at memory location 4097 (decimal) rather than at 1025 as in Commodore's other computers. Thus, if you wish to use features available for some of the other computers, such as Toolkit, to edit or modify a program written or stored from a VIC, you need to relocate the program so it starts at memory location 1025.

Here is a quick and simple method of relocating VIC programs. We will describe two versions, one using the built-in monitor and the other done in direct mode.

To relocate a VIC program from the monitor, load the program from tape and then enter the monitor with SYS1024. Next, look at the first part of BASIC memory by typing M 0400 0400. Make the following changes to the displayed memory:

```
.M 0400 0401
.: 0400 00 01 10 00 00 99 00 XX
```

Next, exit the monitor by typing an X. Now type LIST and the VIC program should list out with an additional line (line 0) at the beginning: 0 PRINT. Finally, type "0" and RETURN. The VIC program is relocated and can be edited or modified at will.

To accomplish the same change in direct mode, the following six POKes are entered:

```
POKE 1025,1:POKE 1026,16  
change link pointers to VIC program  
POKE 1027,0:POKE 1028,0  
create line #0  
POKE 1029,159  
put PRINT on line 0  
POKE 1030,0  
end of line indicator
```

Now, as above, type LIST and the VIC program will list with the additional line 0 PRINT.

Last, type "0" and the line 0 will be eliminated so the VIC program can be edited.

This method works with both BASIC 3.0 and 4.0 Commodore computers and, though it hasn't been tested on other versions, it should work on those as well. It has been successfully used on both 40 and 80 column machines.

If you should wish to relocate several VIC programs in succession, the following assembly language subroutine can be used. It begins at location 926 in the second cassette buffer and can be called by SYS926. To load this program, enter the monitor and type M 039E 03C8 and change the memory as follows:

```
039E A9 00 8D 03 04 8D 04 04  
03A6 8D 06 04 A9 30 8D 6F 02  
03AE A9 01 8D 01 04 A9 10 8D  
03B6 02 04 A9 99 8D 05 04 A9  
03BE 0D 8D 70 02 A9 02 85 9E  
03C6 60 00 00 00 00 00 00 00
```

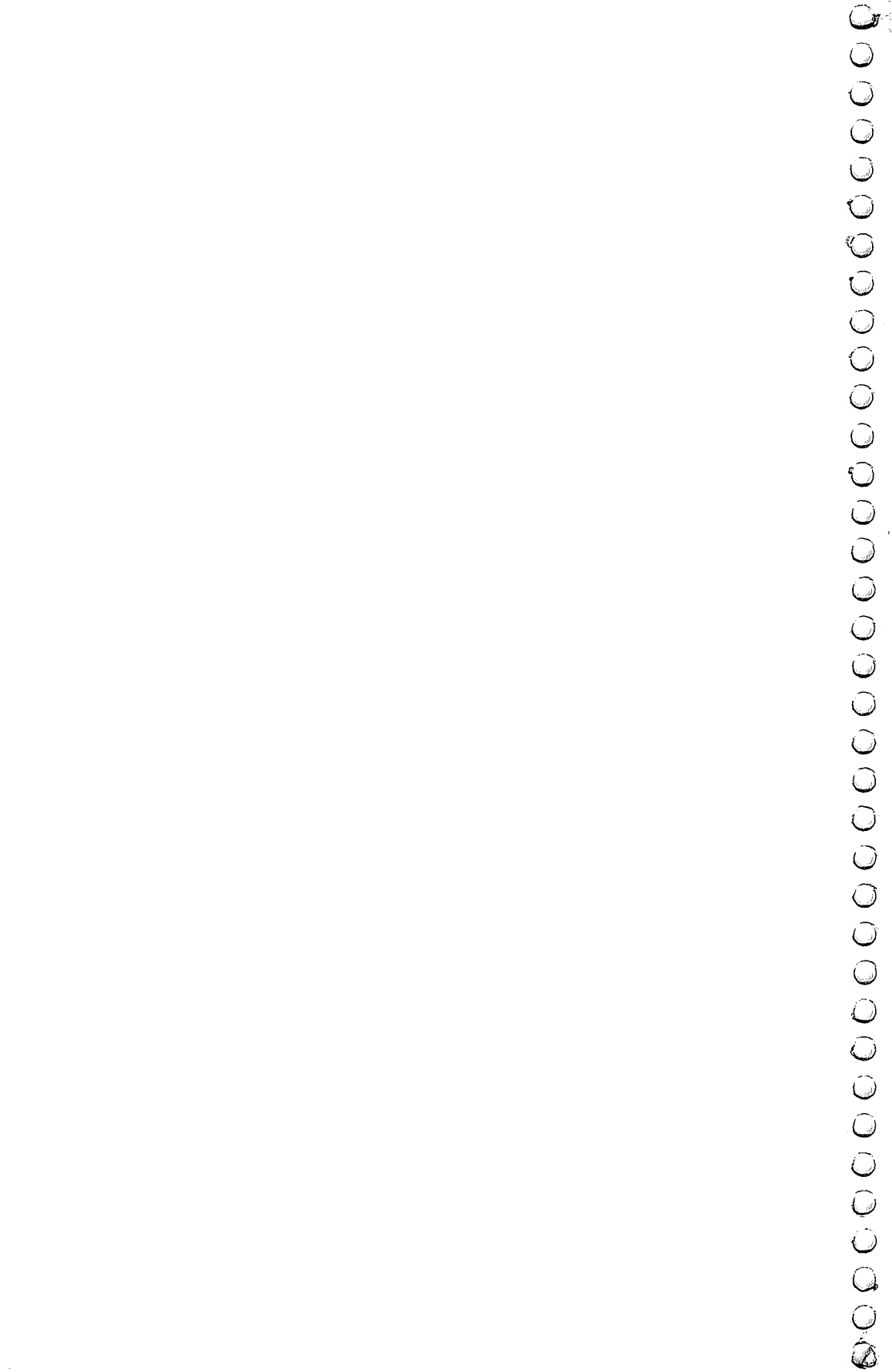
This program can be saved on tape or disk by saving from 039E to 03C8, and then can be loaded in anytime and used to relocate VIC programs with a SYS926 command until the machine is turned off or the second cassette buffer is used for some other purpose. This subroutine is located high enough in the second cassette buffer that disk operations don't overwrite it.

This subroutine automatically erases line 0 so that, when you return to BASIC, the VIC program is moved and ready to be edited or modified without the necessity of removing line 0.

Program 1. Disassembly.

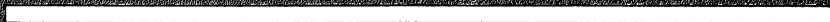
```
039E A9 00 LDA #$00
```

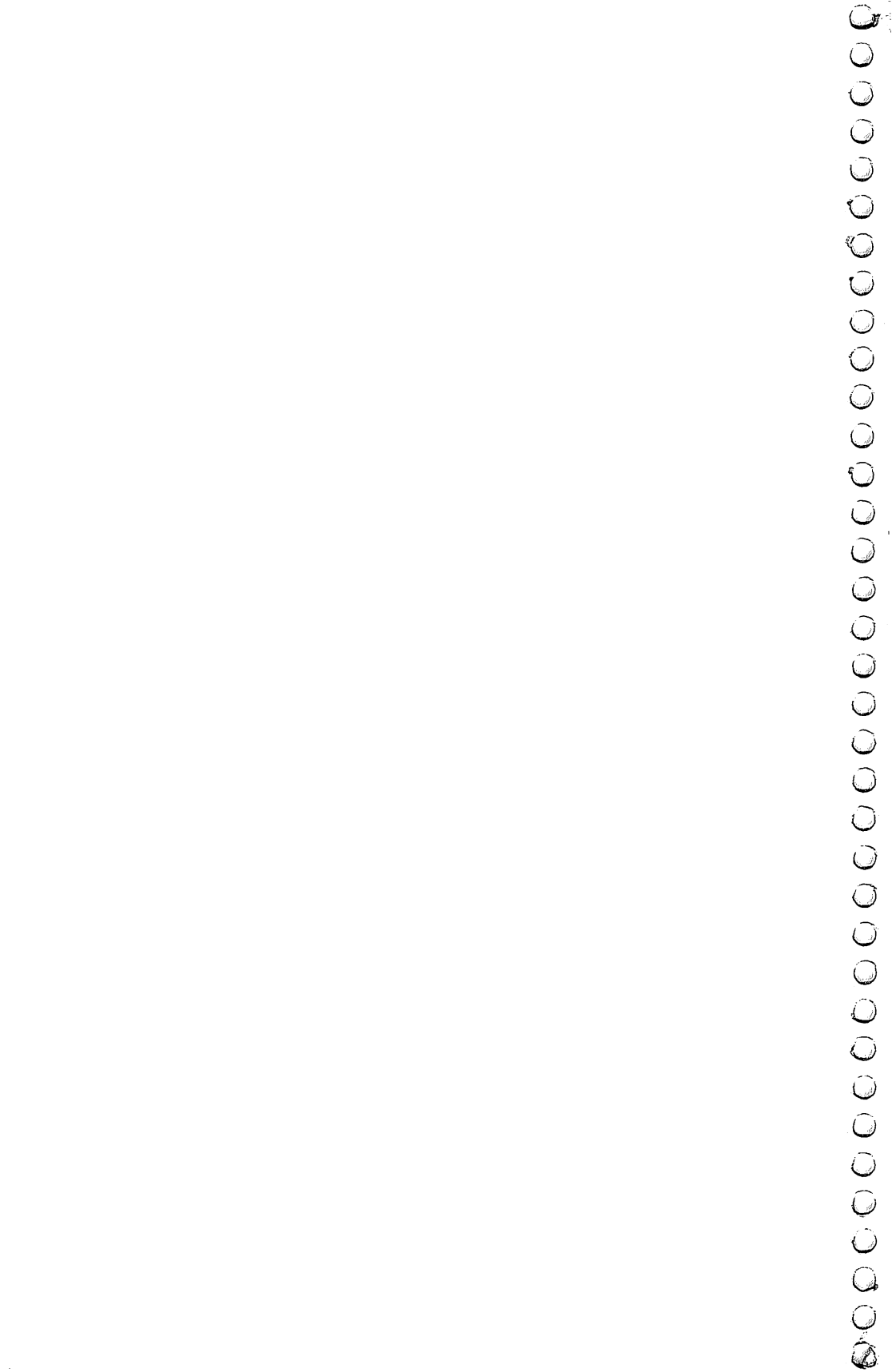
03A0	8D	03	04	STA	\$0403
03A3	8D	04	04	STA	\$0404
03A6	8D	06	04	STA	\$0406
03A9	A9	30		LDA	#\$30
03AB	8D	6F	02	STA	\$026F
03AE	A9	01		LDA	#\$01
03B0	8D	01	04	STA	\$0401
03B3	A9	10		LDA	#\$10
03B5	8D	02	04	STA	\$0402
03B8	A9	99		LDA	#\$99
03BA	8D	05	04	STA	\$0405
03BD	A9	0D		LDA	#\$0D
03BF	8D	70	02	STA	\$0270
03C2	A9	02		LDA	#\$02
03C4	85	9E		STA	\$9E
03C6	60			RTS	
03C7	00			BRK	
03C8	00			BRK	



CHAPTER FOUR

COLOR AND GRAPHICS





Kaleidoscope And Variations

KENNETH KNOX

Here's a short graphics program. After you RUN it, try the interesting variations created by substituting for lines 16 and 70. I like it best after POKE 36879,93 for the screen and border.

Program 1.

```
5 PRINT "{CLEAR}"
10 FOR X = 5 TO 9
15 Z = 7680
16 W = 81
20 POKE (Z+X+22*(X+1)),W
30 POKE (Z+X+22*(X+12)),W
40 POKE (Z+X+11+22*(X+1)),W
50 POKE (Z+X+11+22*(X+12)),W
55 IF Z = 38400 THEN 90
60 IF Z = 7680 THEN 70
70 Z = 38400 : W = 2
80 GOTO 20
90 NEXT X
```

Program 2.

```
16 W = INT(255+10*RND(1))

70 Z = 38400 : W = INT(7*RND(1))
```

High Resolution Plotting

PAUL F. SCHATZ

Follow these guidelines to fully utilize VIC's high resolution graphics potential.

One of the features of the VIC-20 that attracted me was its advertised high resolution color graphics. However, until I acquired *The Programmer's Reference Guide*, I was completely frustrated in trying to implement the hi-res graphics the VIC-20 is ultimately capable of producing.

The book details how to plot a small (64-bit by 64-bit) window on the screen (pp. 88-92). A simple extension of the algorithm in the guide allows creation of a larger (128 by 128) window. Using a memory expansion (either a 3K or 8K), the resolution can be increased to 176 by 160 bits. The methods for creating these windows are detailed in this article. For the basics on high resolution plotting with the VIC-20, the reader should consult *The Programmer's Reference Guide*.

The 128 x 128 Window

To implement high resolution plotting, the screen is filled with programmable characters which are redefined on the fly. Since each character is 8 bits wide by 8 bits high, arranging all 256 characters of the VIC on the screen in a 16-character by 16-character array creates a window with a resolution of 128 bits by 128 bits.

Two hundred fifty-six programmable characters require 2K bytes of RAM (8 bytes per character times 256 characters). With only 3.5K bytes of RAM available in an unexpanded VIC-20, this leaves 1.5K bytes of unallocated RAM. The most efficient arrangement of RAM requires some shuffling so that all 1.5K bytes of RAM are available for BASIC programming. The RAM of the VIC-20 is arranged so that BASIC program and working RAM extends from 4096 to 5631, screen RAM extends from 5632 to 6143, and programmable character RAM extends from 6144 to 8191. The screen is moved using a modification of a method described by Jim Butterfield in "Alternate Screens," reprinted in this book.

Program 1 demonstrates how to set up and draw in the 128 by 128 window. For clarity, versatility, and ease of programming, the program has been divided into several subroutines. These subroutines are located at the beginning of the program for speed and economy of bytes when calling them. The program outlines the window with a black line, draws a horizontal axis, and plots (with black dots) a sine curve. After the graph is finished, pressing any key will change the color of the lines and dots from black to green. After the color has been changed, pressing any key will clear the screen. Pressing any key one more time will restore the screen to the normal display and exit the program.

Program 1.

```

0 POKE52,22:POKE56,22:CLR:GOTO10
1 FORI=6144TO8191:POKEI,0:NEXT:RETURN
2 POKE36864,11:POKE36865,34:POKE36866,144:PO
  KE36867,32:POKE36869,222:POKE36879,25

3 FORI=0TO255:POKE5632+I,I:NEXT:RETURN
4 FORI=0TO255:POKE38400+I,CO:NEXT:RETURN
5 CH=INT(Y/8)*16+INT(X/8)
6 RO=(Y/8-INT(Y/8))*8:BY=6144+8*CH+RO
7 BI=7-(X-INT(X/8)*8):POKEBY,PEEK(BY)OR(2↑BI
  ):RETURN
8 POKE36864,5:POKE36865,25:POKE36866,150:POK
  E36867,46:POKE36869,208:POKE36879,27
9 PRINTCHR$(147):RETURN
10 POKE36869,208:POKE648,22:FORJ=217TO228:POK
  EJ,150:NEXT
11 FORJ=229TO250:POKEJ,151:NEXT
12 GOSUB2:CO=0:GOSUB4:GOSUB1
13 FORX=0TO127:Y=0:GOSUB5:Y=64:GOSUB5:Y=127:G
  OSUB5:NEXTX
14 FORY=0TO127:X=0:GOSUB5:X=127:GOSUB5:NEXTY
19 FORX=0TO127:Y=INT(64+63*SIN(X/10)):GOSUB5:
  NEXTX:A$=""
20 GETA$:IFA$=""THEN20
21 CO=5:GOSUB4:A$=""
22 GETA$:IFA$=""THEN22
23 GOSUB1:A$=""
24 GETA$:IFA$=""THEN24

```

25 GOSUB8
26 END

Here's an explanation of Program 1:

- 0** Moves the end of memory and start of strings pointers and skips over the subroutines to the start of the program.
- 1** Clears the high resolution screen. All the bytes of the character set are zeroed (no pixels are turned on).
- 2-3** Sets up the high resolution screen. POKE 36864 centers the screen vertically, POKE 36865 centers the screen horizontally, POKE 36866 sets a 16-column display ($144 = 128 + 16$), POKE 36867 sets a 16-row display ($32 = 16 * 2$), POKE 36869 defines the start of the programmable character set, and POKE 36879 defines the border and screen colors (25 = white screen and white border). All 256 characters are placed on the screen in a 16 by 16 character array, left to right, top to bottom.
- 4** Defines the color of the points plotted.
- 5-7** Turns on the pixel at coordinates X, Y. The origin ($X = 0, Y = 0$) is the upper left corner of the window. This is a slight modification of the plotting routine given in *The Programmer's Reference Guide*.
- 8-9** Restores the screen to a normal display and clears the screen.
- 10-11** The start of the program. The beginning of the screen RAM is moved to 5632.
- 12** Sets up the high resolution display.
- 13** Draws three horizontal lines (top and bottom borders and horizontal axis).
- 14** Draws two vertical lines (left and right borders).
- 19** Plots the sine curve.
- 21** Changes the color of the plotted points to green.
- 23** Clears the high resolution screen.
- 25** Restores the VIC-20 to the normal screen display.

The 176 x 160 Window

Now the question is "How can resolution better than 128 by 128 be achieved, since the complete set of 256 characters available to the VIC-20 has been used?" The solution lies in the character cell size. Normally characters are 8 bits wide by 8 bits high. This allows for a total of 16384 bits to be plotted (64 bits per character times 256 characters). However, setting a bit in one of the VIC registers (bit 0 of location 36867) changes the character cell size to 8 bits wide by 16 bits high. By redefining the character cell size, theoretically twice as

many points (i.e., 32768) can be plotted. In practice there are factors which limit achieving the maximum. Most important of these is where the programmable character set RAM and screen RAM have to be located. For all practical purposes, these must be located in the RAM addressed from 4096 to 8191. Since 32768 bits require 4096 bytes (32768 bits divided by 8 bits per byte) of storage, the character set and screen will have to overlap. It also becomes obvious that expansion RAM will be required for this higher resolution since all the built-in RAM of the VIC-20 is consumed for screen and character set. Either a 3K or 8K RAM expander can be used. If an 8K expander is used, the start of BASIC has to be moved.

After exploring many different configurations, I finally compromised on a 176 by 160 display. Limiting the display to this size resolves the conflict between the character set and screen, and makes the routines for setting up the display and plotting points the most straightforward. I have opted for the easy route and have left it to the more innovative programmers to push the resolution higher. For a 176 by 160 display, 220 characters, each 8 bits wide by 16 bits high (22 columns by 10 rows), are used. The RAM required to store these characters resides from 4096 to 7615 (220 characters times 16 bytes per character = 3520 bytes). The screen RAM is located from 7680 to 8191. Program 2 is an example of 176 by 160 high resolution plotting on a VIC-20 with 3K RAM expander.

Program 2.

```

0 POKE52,22:POKE56,22:CLR:GOTO10
1 FORI=4096TO7615:POKEI,0:NEXT:RETURN
2 POKE36865,30:POKE36867,21:POKE36869,252:PO
  KE36879,30
3 FORI=0TO219:POKE7680+I,I:NEXT:RETURN
4 FORI=0TO219:POKE38400+I,CO:NEXT:RETURN
5 CH=INT(Y/16)*22+INT(X/8)
6 RO=(Y/16-INT(Y/16))*16:BY=4096+16*CH+RO
7 BI=7-(X-INT(X/8)*8):POKEBY,PEEK(BY)OR(2↑BI
  ):RETURN
8 POKE36865,25:POKE36867,46:POKE36869,240:PO
  KE36879,27
9 PRINTCHR$(147):RETURN
10 REM**START OF PROGRAM
12 GOSUB2:CO=0:GOSUB4:GOSUB1
13 FORX=0TO175:Y=80:GOSUB5:NEXTX

```


CHAPTER FOUR

```
19 FORX=0TO175:Y=INT(80+79*SIN(X/10)):GOSUB5:
   NEXTX:A$=""
20 GETA$:IFA$=""THEN20
21 CO=5:GOSUB4:A$=""
22 GETA$:IFA$=""THEN22
23 GOSUB1:A$=""
24 GETA$:IFA$=""THEN24
25 GOSUB8
26 END
```

Programs 1 and 2 are very similar. Here's where statements in Program 2 differ significantly from those in Program 1:

- 2-3** The initialization subroutine which sets up the high resolution screen. POKE 36865 centers the screen horizontally, POKE 36867 sets a 10 row display of 8 by 16 characters ($21 = 10 * 2 + 1$), POKE 36869 defines the start of the programmable character set, and POKE 36879 defines the border and screen colors (30 = white screen and blue border). Two hundred twenty characters are placed on the screen in a 22 by 10 character array, left to right, top to bottom.
- 5-7** The plotting subroutine which turns on the pixel at coordinates X, Y. The origin (X=0, Y=0) is the upper left corner of the window. Notice that the Y values are divided by 16 instead of 8. Also different is the number of columns (22 replaces 16 as a multiplier in statement 5).
- 13** Draws a horizontal axis in the center of the screen.

To use the high resolution screen with the 8K RAM expander, the start of BASIC must first be moved to 8192 by entering the following command in the direct mode:

POKE44,32:POKE8192,0:NEW

Now enter Program 2 with the following modification and additions:

```
0 GOTO 10
10 POKE 36866,150:POKE 36869,240:POKE 648,30
11 FORJ=217TO228:POKEJ,158:NEXT:FORJ=229TO250
   :POKEJ,159:NEXT
```

The POKES to set the end of memory and start of strings are no longer needed since the screen RAM is located before the BASIC RAM. Plugging the 8K RAM expander into the VIC-20 automatically moves the start of screen RAM to 4096. Statements 10 and 11 move

the start of screen RAM back to 7680.

Reference:

A. Finkel, N. Harris, P. Higginbottom, and M. Tomczyk, VIC-20 Programmer's Reference Guide, Commodore Business Machines, Inc. and Howard W. Sams & Co., Inc., 1982.

VIC Color Tips

CHARLES BRANNON

Users of other computers, such as the Atari or Apple, will find the VIC harder to use for color graphics because there are no dedicated statements for controlling these features. For first time users, this should make things easier.

The only command that can be used for graphics besides PRINT is POKE. POKE places a number into a memory location. Its format is POKE A,B. A is the memory location, and B is the value to be placed there, zero to 255. Some spots in memory can control Input/Output chips, such as the Video Interface Chip inside the VIC. Location 36879 is the control register for background and border colors. To get each combination, you place a number from zero to 255 into 36879, as previously mentioned. For any particular combination, you can look up the colors in the table at the end of this article (Table 2). There is an easier way, however, at least from a programming standpoint.

An Easier Way

The DEF FN command allows the programmer to design his own function. The VIC has, for example, the standard INT function. INT(X) will give you the whole-number value of the argument X by dropping the fractional portion. It does not round X. To provide a rounding-up function, we can use the DEF FN command. To round dollar and cents amounts, the statement $\text{DEF FNR}(V) = \text{INT}(V * 100 + .5) / 100$ is executed at the start of the program. After that, FNR(X) will give you the rounded version of X, or any value in parentheses. PRINT FNR(3.1415927) will return 3.14, while PRINT FNR(500.076) will give 500.08. The R after the FN is a label to remind you what the function does. Here R stands for Round. These labels have the same format as numerical variable names.

What we want to do is to devise a formula which will give us the right number from the table for each color, one to 16. We will give the background color from one to 16 through the FN routine, and it will give us the number ready for POKEing. To get any background color from any of the 16 possible colors, just multiply the color number by 16 and then subtract eight. We can code this as

DEF FNC(V)=V*16-8. Remember, V is just a dummy variable used to define the relationship of the argument (what we give the routine) in the formula. Next we use a little shorthand. The number 36879 (the color control) is a little hard to remember, and it does not look much different than any other memory location. We will make it easier to remember (make it *mnemonic*) by making it a variable, SCREEN=36879. Now we can call forth any of our 16 colors with the statement: POKE SCREEN, FNC (*color*), where *color* is the number from one to 16. This almost looks like a real graphics command.

Adding Border Colors

What about the border colors? In addition to the background, you can have eight border colors, numbered from zero to seven. This is one less than the corresponding number on the color keys (CTRL-6 would be 5). Now just take this number and add it to the number that you POKE into SCREEN. Now we just use: POKE SCREEN, FNC (*color*) + *border*, where *border* is the border color, zero to seven. If you don't use border colors, or don't add anything to FNC(*color*), then the border will be black.

Remember that if the background is the same color as the text, the cursor will become invisible. If you need to, set things straight with POKE 36879,27 or hold down RUN/STOP and press RESTORE to reset.

The little program at the end of this article demonstrates what I've been talking about by displaying all the combinations of screen and border colors. It's simple to figure out, so look it over and get to work on your VICtorious applications!

Table 1. Screen/Border Colors.

<u>Screen</u>		<u>Border</u>
1 Black	12 Light Cyan	0 Black
2 White	13 Light Purple	1 White
3 Red	14 Light Green	2 Red
4 Cyan	15 Light Blue	3 Cyan
5 Purple	16 Light Yellow	4 Purple
6 Green		5 Green
7 Blue		6 Blue
8 Yellow		7 Yellow
9 Orange		
10 Light Orange		
11 Pink		

CHAPTER FOUR

```
100 REM * ANOTHER RAINBOW *
110 DEF FNC(V)=V*16-8
120 SCREEN=36879
130 FOR BK=1 TO 16
140 PRINT "{CLEAR}{WHT}";
150 IF BK>1 THEN PRINT "{BLK}";
160   PRINT "SCREEN";BK
170   FOR BD=0 TO 7
180     POKE SCREEN,FNC(BK)+BD
190     PRINT,"BORDER";BD
200     FOR W=1 TO 500:NEXT W
210   NEXT BD
220 NEXT BK
230 POKE SCREEN,27
240 END
```

Table 2. POKE Values.

	BACKGROUND #								BORDER							
	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1:	8	9	10	11	12	13	14	15								
2:	24	25	26	27	28	29	30	31								
3:	40	41	42	43	44	45	46	47								
4:	56	57	58	59	60	61	62	63								
5:	72	73	74	75	76	77	78	79								
6:	88	89	90	91	92	93	94	95								
7:	104	105	106	107	108	109	110	111								
8:	120	121	122	123	124	125	126	127								
9:	136	137	138	139	140	141	142	143								
10:	152	153	154	155	156	157	158	159								
11:	168	169	170	171	172	173	174	175								
12:	184	185	186	187	188	189	190	191								
13:	200	201	202	203	204	205	206	207								
14:	216	217	218	219	220	221	222	223								
15:	232	233	234	235	236	237	238	239								
16:	248	249	250	251	252	253	254	255								

The Window

CHARLES BRANNON

The VIC's window to the world is your TV set. You can control the window with PEEKs and POKEs.

Think about it. What is involved in placing a character on the screen? If you look at one closely, you can see that each character is made up of tiny dots. (Try SHIFT-M.) These dots are the smallest images the VIC can produce. They are called *picture elements*, or simply *pixels*. If you have the VIC Super Expander Cartridge, you should be able to put any individual pixel anywhere on the screen. For text and keyboard graphics, though, the VIC handles all that for you.

You may have heard that the screen is stored somewhere in memory. Why? And perhaps more perplexing – how? If you do not already know how images are formed on a television screen, here is a brief explanation. Inside the picture tube is a small rod called an *electron gun* which “shoots” a fast stream of electrons, much as a machine gun shoots bullets. These electron bullets strike a special coating on the inner front surface of the screen. After a spot is hit by the beam, it will glow for only a brief time, and then fade away. Therefore, you cannot just smack something on the screen and expect it to stay there. It is necessary to constantly re-draw the image on the TV screen. The VIC *refreshes* the screen sixty times a second. In order to do that, the intended contents of the screen must be stored somehow so that they can be constantly fetched and placed on the screen.

There is another reason why characters are stored in memory. That way, you can look at that memory and effectively let the computer “look” at its own screen. You can also make characters appear on the TV screen as if by magic, merely by changing some of that memory. So, the numbers in the computer's memory are placed in a pattern of light and dark spots on the screen to form images. In effect the TV screen is a window looking at the memory set aside for it.

Now we are ready to tackle the next question. How is this done? As mentioned, each character is made of pixels. All of the 255 characters that the VIC can print are eight pixels long and eight pixels wide. So, each box on the screen could contain up to 64 pixels

(RVS-space uses them all). There are 23 rows of 22 characters on the VIC's screen. Therefore, there are 506 boxes, and 506 x 64 pixels. This makes an incredible 32,384 pixels for the belabored VIC to keep track of.

What Color Are Characters?

Fortunately, information is not stored on the screen pixel by pixel. In reality, each of the 506 boxes corresponds to a memory location. A memory location can hold a number between zero and 255, and the VIC can display up to 255 characters. Voilà! A perfect match! And, indeed, that is the way it works. Each character is assigned a number from zero to 255. The numbers 1-26 refer to the letters A-Z. That's easy enough. Numbers are 48-54 (0-9).

The only other thing you need to know in order to tickle your TV is how the VIC keeps track of what color each character is. There are another 506 memory locations (boxes, if you like) to correspond to the color of a character in that box. Each memory box holds a number from zero to seven. These are the numbers that are above each "color control key," minus one. Because there are two separate areas for text and colors, you can change one without altering the other. You can change the color of selected screen boxes, or change the letters, numbers, or graphics in a box, yet keep the same color.

Now we are ready to get down to business. The text memory starts at 7680 and naturally ends at 8185. (Note: To allow for "floating memory" on VICs with more than 4K of RAM, use: $\text{TEXT} = 4 * (\text{PEEK}(36866) \text{ AND } 128) + 64 * (\text{PEEK}(36869) \text{ AND } 120)$; $\text{COLOUR} = 37888 + 4 * (\text{PEEK}(36866) \text{ AND } 128)$. See **COMPUTE!**, October 1981, #17.) The color memory begins at 38400. To make things easier, we'll use full variable names to label these numbers. $\text{TEXT} = 7680$ and $\text{COLOUR} = 38400$. Why the British spelling of COLOR? Otherwise the VIC would see it as $\text{COL OR} = 38400$. "OR" is a *reserved word* (it's used by BASIC), so it would give a nasty SYNTAX ERROR.

Now, in order to find the location of each box, we'll use the simple formula: $\text{LOC} = \text{CLM} + 22 * \text{ROW}$. ROW is the line, or vertical part, and CLM is the column, or the horizontal component. LOC will now contain a number from zero to 505, assuming that you number rows from zero to 22 and columns from zero to 21, and that is just what you have to do. Okay, now let's pick our character (say 102, the gray square) and get POKeIng. That's right, we use the command POKE to place our numbers into memory.

POKE is used in this format: `POKE mem,num` where *mem* is the

memory location from zero to 65535, and *num* is the number you want to put there from zero to 255. Remember, be careful with POKE. You have to remember that the VIC uses some memory for its own secret things, and it might temporarily go crazy (hang up) if you confuse it by POKEing randomly.

On your mark, get set... POKE! We use the following command, assuming that all our variables have been defined: POKE TEXT + LOC, 102:POKE COLOUR + LOC,5. This makes a gray square appear on the screen at the predefined location. But it's not really gray. It's more like apple-green. This is a nifty tip: If you only want to draw with solid squares, you can have 16 colors: eight from reverse-field spaces, and eight lighter-colored ones from "grey squares" (what else could we call them?).

Now you have the basis for using your television window. You take it from there. You can design some remarkable application for it (like coordinate plotting, or bar graphs).

```

1000 REM      *** BORDER ***
1010 REM FOR USE AS A SUBROUTINE
1020 REM GOSUB 1000, OR
1030 REM GOSUB 1080 TO CALL
1040 REM WITHOUT CLEARING THE
1050 REM SCREEN
1060 PRINT "{CLEAR}"
1070 DEF FNR(V)=INT(7*RND(1)+1)
1080 SCREEN=36879:POKE SCREEN,8:REM BLACK
1090 TEXT=4*(PEEK(36866)AND128)+64*(PEEK(36
      869)AND120)
1100 COLOUR=37888+4*(PEEK(36866)AND128)
1110 FOR I=0 TO 22
1120 POKE TEXT+I,102
1130 POKE TEXT+22*22+I,102
1140 POKE TEXT+22*I,102
1150 POKE TEXT+21+22*I,102
1160 POKE COLOUR+I,FNR(0)
1170 POKE COLOUR+22*22+I,FNR(0)
1180 POKE COLOUR+22*I,FNR(0)
1190 POKE COLOUR+21+22*I,FNR(0)
1200 NEXT I
1210 RETURN

```


Custom Characters For The VIC

DAVID MALMBERG

Custom characters permit a programmer to easily display and move multicolored “shapes” for fast and simple graphics, or specialized alphabets for foreign languages.

One of the many innovations built into the new Commodore VIC is the ability to design our own special characters and have them available to BASIC programs. The possible uses are many. Now you can have different language fonts, such as Japanese, Chinese, or Arabic. You can display electronic schematic symbols. Or, you can write a program that transcribes stenographic characters into English. Greek alphabet characters are now available for tutorial math programs requiring special symbols. You can even design your very own *Space Invaders* creatures.

This article explains how to make these custom characters. It also presents a utility program to make the job of designing these characters and incorporating them into your BASIC programs quite easy and straightforward. Finally, a sample program is given that demonstrates the custom character features of the VIC by displaying all of the special math symbols of the Greek alphabet.

VIC Character Sets

The character set to be used by the VIC is determined by the value in location 36869. [Note: all locations are given in decimal.] This is similar to location 59468 in the PET or CBM machines. The various VIC character sets specified by POKEing this location are as follows:

POKE36869,240 gives uppercase and graphics (when shifted)
POKE36869,242 gives lowercase and uppercase (when shifted)
POKE36869,255 causes the VIC to set aside the first 64 characters of the character set as user-defined characters. These special characters will be determined by values in the 512 locations beginning at 7168.

VIC Character Representation

To understand how to design your own VIC characters, you must first understand how the VIC represents its characters internally. Just how this is done was demonstrated by Jim Butterfield in his article in the April 1981 issue of **COMPUTE!**. Jim pointed out that the two "normal" character sets can be located by using the following equation:

$$\text{CHR}(I) = 32768 + B + 8 * I$$

where **B = 0** for the upper/graphics set

and **B = 2048** for lower/upper characters

and **I = the "screen POKE" value of the character**

(e.g., @ is 0, A is 1, B is 2, etc.)

As an example, let's look at how the VIC stores an uppercase "A." It has a "screen POKE" value of 1, so by using the above equation we see that it is stored in the eight consecutive bytes beginning at location 32776. If we were to PEEK these locations, we would find the following decimal values – which have the specific bit patterns which define the pixel (i.e., dot) pattern the VIC uses when it prints an "A." The bit pattern corresponds to the binary representation of the decimal number found in the location. For example, location 32782 contains a decimal 66 which is 01000010 in binary – i.e., the pattern at the bottom of the VIC's representation of an "A".

BYTE LOCATION	DECIMAL VALUE	BIT PATTERN							
		7	6	5	4	3	2	1	0
32776	24				*	*			
32777	36		*				*		
32778	66	*							*
32779	126	*	*	*	*	*	*	*	*
32780	66	*							*
32781	66	*							*
32782	66	*							*
32783	0								

Defining Your Own Characters

Let's see how you would go about designing and incorporating your own custom character into a BASIC program. For example, let's add the following vicious-looking creature to your version of *Space Invaders* or *Dunjonquest*.

CHAPTER FOUR

	PIXEL PATTERN								BINARY	DECIMAL
	7	6	5	4	3	2	1	0		
ROW0	*							*	10000001	129
ROW1	*			*	*			*	10011001	153
ROW2		*	*			*	*		01100110	102
ROW3			*	*	*	*			00111100	60
ROW4	*	*	*	*	*	*	*	*	11111111	255
ROW5			*	*	*	*			00111100	60
ROW6		*						*	01000010	66
ROW7		*						*	01000010	66

The binary and decimal values corresponding to the creature's pixel pattern are also given. To get the VIC to use this pattern as one of its characters, let's enter and run the following short program:

```

100 X = PEEK(56)-2:POKE 52,X:POKE 56,X:POKE 51,PEEK
    (55):CLR
110 CS = 256*PEEK(52) + PEEK(51)
120 FOR I = CS TO CS + 511:POKE I,PEEK(I + 32768-CS):NEXT
130 FOR I = 0 TO 7:READ J:POKE CS + I,J:NEXT
140 DATA 129,153,102,60,255,60,66,66
150 POKE 36869,255:PRINT"CLR"
160 FOR I = 1 TO 11:PRINT"@":NEXT

```

After running the program you should see a row of 11 of your creatures on the top line of the screen.

Let's review this program line-by-line to understand how to use these special characters in other programs. Line 100 PEEKs two pages lower (a page is 256 bytes). Line 100 also changes the pointer to the beginning of the string variables (locations 51 and 52) to point to the beginning of these two pages. The CLR resets the user RAM boundaries so that these two pages are protected from the rest of the BASIC program. Line 110 calculates the starting location for the table containing the new character set.

Line 120 transfers the first 65 characters of the standard uppercase character set from ROM into the new character set table in the top two pages of user BASIC RAM. This is not strictly required, but it is good practice because it allows you to have access to "normal" characters as well as your specially designed characters on the same screen.

Line 130 reads the data in line 140 that defines the pixel pattern for the creature and POKEs it into the table space used by the first character of the new character set, i.e., the table space used by the "@" sign in the normal uppercase character set.

Line 150 tells the VIC to use the custom character set where the first 64 characters are user defined. Line 160 tries to print a row of @'s but ends up printing your creature because its pixel pattern is in the table where the @ would normally be located.

You could continue to add to this simple program to build a complex game that would use your creature whenever you PRINTed "@" or POKEd the screen with a zero (i.e., the @'s normal screen POKE value). To return to the normal character set, give the direct command: POKE 36869,240 which will cause all your creatures to be transformed back to @'s.

A Utility Program

Program 1 is a short BASIC program for the VIC which helps with the designing, testing, and coding of special characters by essentially automating the process described above. The program has two operation modes: (1) a review mode which allows you to see how the current character set looks – including your custom characters, and (2) an editing and new-character definition mode.

Review Mode

When you first run Program 1, you will initially be in the review mode and the screen will look like this:

@ABCDEFGH	OPTIONS
IJKLMNOP	
QRSTUVWXYZ	NEW CHAR
XYZ[£]	EDIT CHAR
■! " # \$ % & ' () * + , - . /	QUIT
01234567	
89::(: = } ?	

The characters shown in the first eight rows and eight columns of the screen are the currently defined custom character set. Note that you start the program with the normal uppercase characters. As you redefine the characters, the new characters will be displayed in their appropriate place in this character table. For example, if we had used the utility program to create the creature in the previous example, it would be displayed in place of the @ sign whenever we were in the review mode.

The blank in the first column of the fifth row will be red and will serve as a "fake" cursor. You will be able to use all normal cursor controls, including HOME and CLR, to position this fake cursor on any of the characters displayed in the character set. This fake cursor

will also have automatic repeat key and automatic wraparound features.

To define a new special character, move the fake cursor to the position of the character in the normal character set you wish to replace. A good idea is to replace characters that are seldom used, so that you still have access to the more popular characters, i.e., the letters and digits. Once the cursor is positioned, hit "N" on the keyboard to define a new character in place of the one the cursor is on. If you are just reviewing a character that you have previously created and decide it needs more work, then position the cursor on the character and hit "E". Either "E" or "N" will shift the program into the EDIT mode.

Edit Mode

As an example, let's assume that you wanted to add serifs to the character "K". After placing the red cursor over the K, you would hit an "E" to enter the Edit mode, and the screen would look like this:

■.*...*

..*.*

..*.*

..*.*

..*.*

..*.*

..*.*

..*.*

.....

OPTIONS

+ ADD DOT

- ERASE

= UPDATE

B BASIC

R REVIEW

Q QUIT

The screen shows the pixel pattern for the character K in a large eight-by-eight format. The cursor is "homed," but may be re-positioned using the cursor control keys to rest on any of the large pixels. Once the cursor is properly positioned, a pixel may be "turned on" by hitting a "+" or "turned off" by hitting a "-" sign. After you are satisfied with your handiwork, hit an "=" sign to put that character into the character set table. Then if you wish to see the character in its normal size and format, hit an "R" to go back to the Review mode.

After the design of the special character is complete, you may hit a "B" to have the VIC print the BASIC code needed to add this character to other programs. For example, if we had used this utility program to design the creature used in the previous example, and we hit a "B", the VIC would display the following lines of BASIC code:

```
200 READ X:FOR I=X TO X+7:READ Y:POKE X,Y:NEXT
210 DATA 7168, 129, 153, 102, 60, 66,66
```

You will recognize that these lines of code are essentially equivalent to lines 130-140 in the previous example. IMPORTANT – these lines of code will work only if lines 100-120 in the previous example or their equivalent have already been executed. Program 2 gives an example of how the BASIC code generated by this utility program might be used to incorporate a number of special characters (specifically, the math symbols in the Greek alphabet) into a BASIC program.

A word of caution – hitting either “B” or “R” will generate displays based on what is actually in the character set data table. This may not correspond to the current large-sized pixel pattern. Always be sure this table is correct by updating it via the “=” command prior to using the “B” or “R” commands.

Hitting a “Q” while in either the Edit or Review modes will cause both the memory size and the character set to be reset to their normal states and the program to end.

A Few Suggestions

If you want to use a custom pattern that is larger than just one character, use the utility program to design the pieces of the overall pattern into contiguous characters, as shown in the Review mode display. For example, if you want a three-across by two-down pattern, you could use the utility program to design the various parts into the character positions normally occupied by @, A, B, and H, I, J. Then whenever you PRINTed these characters in your BASIC program (in the correct configuration – of course), you would get your desired large custom pattern.

If the reverse character flag is on (i.e., the character you are PRINTing has been preceded by a reversed “R”), the VIC will use the standard character set and not the custom character set. You will find this useful when you have already redefined various characters, and you want to use those same original characters on the same screen. You can simply PRINT those characters in reverse. This “trick” is used in the Review mode of the utility program to assure that the options are always printed properly.

Programming Challenges

Here are two challenges to programmers who would like to show they have mastered the VIC’s custom character features and who want to write very useful programs that can be used and enjoyed by the growing VIC community: (1) Write a program that will draw a straight line (or as close to one as possible) between any two pixels on the VIC’s screen. (2) Write a generalized graph program that can

CHAPTER FOUR

graph equations (one or more simultaneously) in high resolution by defining special characters – on the fly – as required by the shape of the equations.

Program 1.

```
100 POKE 36879,27:PRINT "{CLEAR} CHARACTER GEN
    ERATOR"
110 PRINT"{02 DOWN} BY DAVID MALMBERG"
120 REM 43064 VIA MORAGA
130 REM FREMONT, CALIFORNIA
140 X=PEEK(56)-2:POKE52,X:POKE56,X:POKE51,PEEK
    (55):CLR
150 CS=256*PEEK(52)+PEEK(51)
160 FORI=CSTOCS+511:POKEI,PEEK(I+32768-CS):NEX
    T
170 S=7680:CL=22
180 CR=0:LN=200:P=12:BG=3:BR=1
190 POKE36879,BG*16+BR
200 DEFFNA(XX)=S+R*CL+C:REM SCREEN POKE LOCATI
    ON
210 DEFFNB(XX)=8*R+C:REM SCREEN POKE VALUE FOR
    CHARACTER
220 GOTO580
230 PRINT"{CLEAR}":GOSUB810
240 PRINT"{HOME}";:FORI=0TO7:PRINT".....":N
    EXT:F=0
250 PRINT"{HOME}":R=0:C=0
260 Z=FNA(0)
270 IFF=0THENPOKEZ,PEEK(Z)+128:GOTO310
280 IFZ=ZLTHEN300
290 POKEZL,IL:POKEZL+30720,BC:ZL=Z:IL=PEEK(ZL)

300 POKEZ,32:POKEZ+30720,2
310 GETA$:IFA$=""THEN310
320 IFF=0THENPOKEZ,PEEK(Z)-128
330 REM CURSOR CONTROL OPTIONS
340 IFA$="Q"THENPOKE56,PEEK(56)+2:POKE36869,24
    0:PRINT"{CLEAR}":END
350 IFA$="{RIGHT}"ANDC=7THENC=0:GOTO260
360 IFA$="{RIGHT}"THENC=C+1:GOTO260
370 IFA$="{LEFT}"ANDC=0THENC=7:GOTO260
380 IFA$="{LEFT}"THENC=C-1:GOTO260
390 IFA$="{DOWN}"ANDR=7THENR=0:GOTO260
```

```

400 IFA$="{DOWN}"THENR=R+1:GOTO260
410 IFA$="{UP}"ANDR=0THENR=7:GOTO260
420 IFA$="{UP}"THENR=R-1:GOTO260
430 IFA$="{HOME}"THEN250
440 IFF=1THEN540
450 REM DEFINE NEW CHARACTER OPTIONS
460 IFA$="+ "THENPOKEZ,81:GOTO260
470 IFA$="- "THENPOKEZ,46:GOTO260
480 IFA$=" "THEN680
490 IFA$="{CLEAR}"THEN240
500 IFA$="R"THEN580
510 IFA$="B"THEN770
520 GOTO260
530 REM REVIEW CHARACTER SET OPTIONS
540 CR=FNB(0)
550 IFA$="N"THENPOKE36869,240:GOTO230
560 IFA$="E"THENPOKE36869,240:F=0:GOTO730
570 GOTO260
580 POKE36869,255:R=4:C=0:ZL=FNA(0):IL=32
590 PRINT"{CLEAR}@ABCDEFGH":PRINT"HIJKLMNOP":PRI
    NT"PQRSTUVWXYZ":PRINT"XYZ[\]↑_":F=1
600 PRINT"!"+CHR$(34)+"#$$&'":PRINT"()*+,-./"
    :PRINT"01234567":PRINT"89:;<=>?"
610 PRINT"{HOME}";SPC(12);"{REV}OPTIONS{OFF}":
    PRINT
620 PRINTSPC(10);"{REV}N NEW CHAR{OFF}"
630 PRINTSPC(10);"{REV}E EDIT CHAR{OFF}"
640 PRINTSPC(10);"{REV}Q QUIT{OFF}"
650 BC=PEEK(38400)
660 GOTO260
670 REM UPDATE CHARACTER DATA IN TABLE
680 PRINT"{HOME}";:X=CS+8*CR:FORR=0TO7:SM=0:FO
    RC=0TO7:D=7-C
690 SM=SM-2↑D*(PEEK(FNA(0))=81):NEXTC
700 POKEX+R,SM:PRINTSPC(8);SM:NEXTR
710 R=0:C=0:GOTO260
720 REM EDIT CHARACTER FROM TABLE
730 X=CS+8*CR:PRINT"{CLEAR}":FORR=0TO7:Y=PEEK(
    X+R):FORC=0TO7:Z=FNA(0)
740 Q=46:Y=Y*2:IFY>255THENQ=81:Y=Y-256
750 POKEZ,Q:NEXTC,R:R=0:C=0:GOSUB810:GOTO260
760 REM BASIC STATEMENTS TO DEFINE CHARACTER
770 X=CS+8*CR:PRINT"{HOME}">{08 DOWN}"
780 PRINTLN;"READ X:FOR I=X TO X+7: READ Y: PO
    KE X,Y: NEXT":LN=LN+10

```

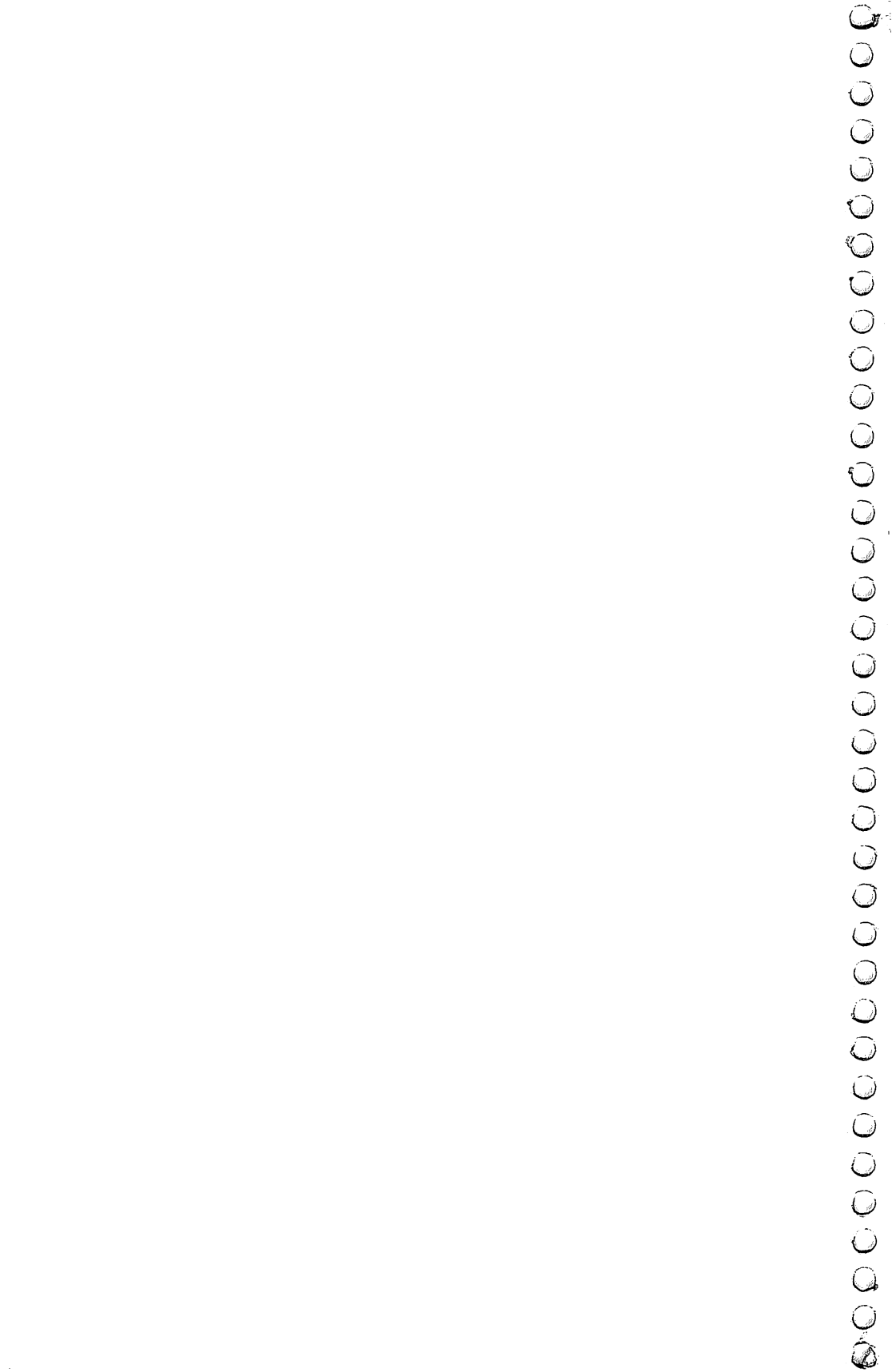

CHAPTER FOUR

```
790 PRINTLN;"DATA";X;:FORI=XTOX+7:PRINT"{LEFT}
  ,";PEEK(I);:NEXTI:PRINT
800 GOTO260
810 PRINT"{HOME}";SPC(13)"{REV}OPTIONS{OFF}":P
  RINT
820 PRINTSPC(P);"{REV}+{OFF} ADD DOT"
830 PRINTSPC(P);"{REV}-{OFF} ERASE"
840 PRINTSPC(P);"{REV}={OFF} UPDATE"
850 PRINTSPC(P);"{REV}B{OFF} BASIC"
860 PRINTSPC(P);"{REV}R{OFF} REVIEW"
870 PRINTSPC(P);"{REV}Q{OFF} QUIT"
880 RETURN
```

Program 2.

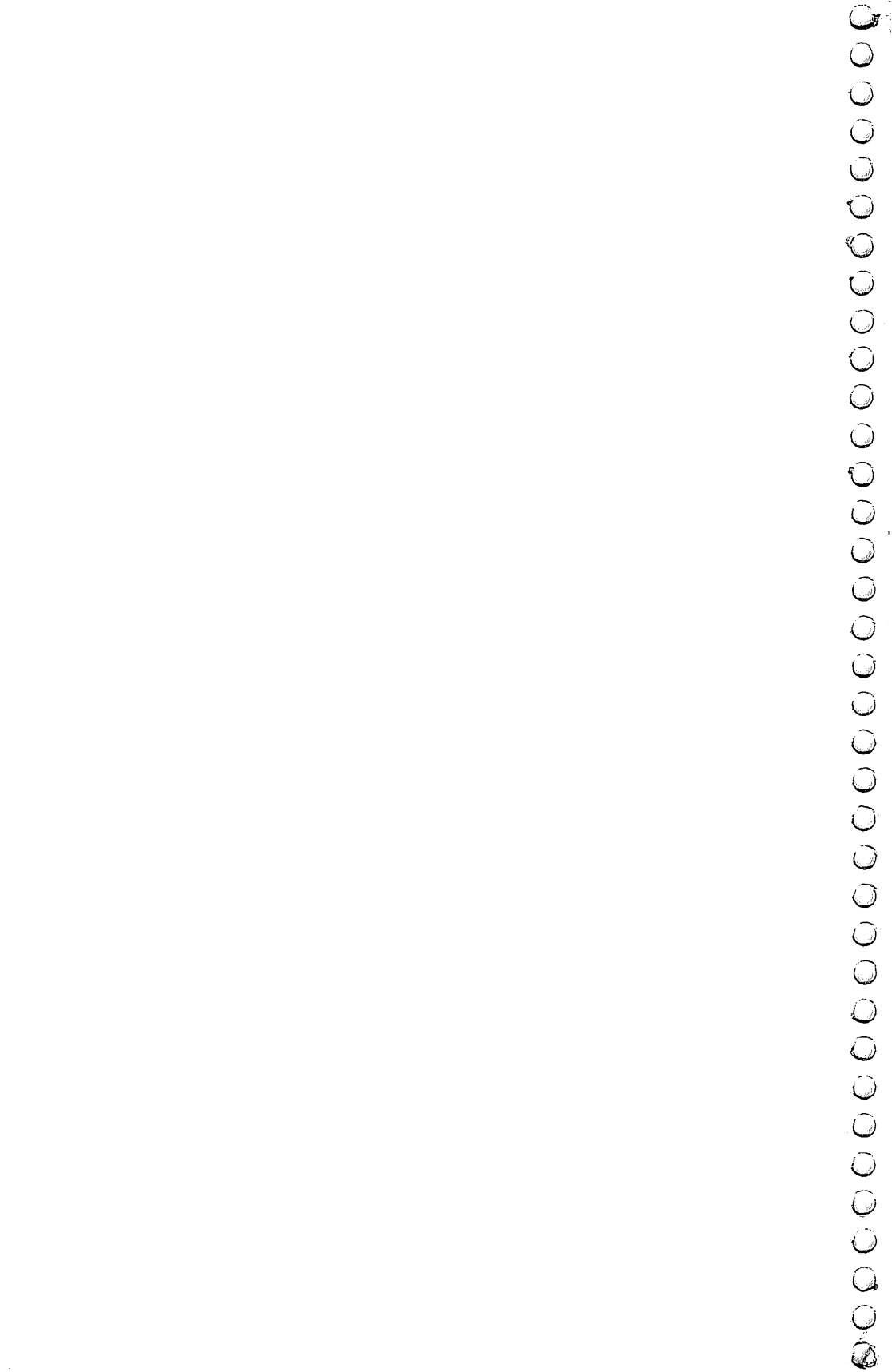
```
100 POKE36879,27:PRINT"{CLEAR} VIC CHARACTER D
  EMO
110 PRINT"{02 DOWN} BY DAVID MALMBERG
120 REM 43064 VIA MORAGA
130 REM FREMONT, CALIFORNIA
140 X=PEEK(56)-2:POKE52,X:POKE56,X:POKE51,PEEK
  (55):CLR
150 CS=256*PEEK(52)+PEEK(51)
160 FORI=CSTOCS+511:POKEI,PEEK(I+32768-CS):NEXT
170 READX:IFX<0THEN200
180 FORI=XTOX+7:READJ:POKEI,J:NEXT
190 GOTO170
200 PRINT"{CLEAR}@ABCDEFGH":PRINT"HIJKLMNO":PRI
  NT"PQRSTUVWXYZ":PRINT"XYZ[\]↑_"
210 PRINT"!"+CHR$(34)+"#$%&'":PRINT"()*+,-./"
  :PRINT"01234567":PRINT"89:;<=>?"
220 PRINT"{HOME}";SPC(11);"{REV}OPTIONS{OFF}":
  PRINT
230 PRINTSPC(11);"{REV}N NORMAL{OFF}"
240 PRINTSPC(11);"{REV}L LOWER{OFF}"
250 PRINTSPC(11);"{REV}G GREEK{OFF}"
260 PRINTSPC(11);"{REV}Q QUIT{OFF}"
270 PRINT:PRINT:PRINT
280 GETA$:IFA$=""THEN280
290 IFA$="N"THENPOKE36869,240
300 IFA$="L"THENPOKE36869,242
310 IFA$="G"THENPOKE36869,255
320 IFA$="Q"THENPOKE36869,240:POKE56,PEEK(56)+
```

```
2:END
330 GOTO280
340 DATA7168,24,24,36,60,102,66,66,0
350 DATA7176,124,34,34,60,34,34,124,0
360 DATA7184,126,34,34,32,32,32,112,0
370 DATA7192,24,24,36,36,102,66,126,0
380 DATA7200,126,34,32,56,32,34,126,0
390 DATA7208,126,70,12,24,48,98,126,0
400 DATA7216,102,36,36,60,36,36,102,0
410 DATA7224,24,36,66,126,66,36,24,0
420 DATA7232,28,8,8,8,8,8,28,0
430 DATA7240,102,36,40,48,40,36,102,0
440 DATA7248,24,24,60,36,36,102,102,0
450 DATA7256,66,102,90,66,66,66,66,0
460 DATA7264,66,98,82,74,70,66,60,0
470 DATA7272,126,0,36,60,36,0,126,0
480 DATA7280,24,36,66,66,66,36,24,0
490 DATA7288,126,36,36,36,36,36,36,0
500 DATA7296,124,34,34,60,32,32,112,0
510 DATA7304,126,98,48,24,48,98,126,0
520 DATA7312,62,42,8,8,8,8,28,0
530 DATA7320,20,42,8,8,8,8,28,0
540 DATA7328,8,28,42,42,28,8,8,0
550 DATA7336,102,66,36,24,36,66,102,0
560 DATA7344,42,42,42,28,8,8,28,0
570 DATA7352,0,24,36,66,66,36,102,0
580 DATA7360,0,0,0,0,0,0,0,0
590 DATA7368,0,0,0,0,0,0,0,0
600 DATA7376,0,0,0,0,0,0,0,0
610 DATA-1
```



CHAPTER FIVE

MAPS AND SPECIFICATIONS



How To Use The 6560 Video Interface Chip

DALE GILBERT

These are the details of the Video Interface Chip (VIC) which controls the video and sound effects.

The 6560 Video Interface Chip, the VIC chip, provides low cost, high resolution, color video for a color monitor or a color television, and it also incorporates a sound generator, A/D converters [*analog/digital*], and even a light pen feature.

The 6560, some RAM, a crystal, a few bus drivers, and a little decode logic is all the hardware that is required to add color and sound to a microprocessor that has an expansion bus.

The VIC capabilities include on-chip sync generation, screen grid size of up to 192 horizontal dots by 200 vertical dots, two character sizes, three independent programmable tone generators, a white noise generator, an amplitude modulator, screen centering, on-chip DMA address generation, and two modes of color operation.

The 6560 VIC is manufactured by MOS Technology, Inc.. Commodore Business Machines incorporates the 6560 in its VIC-20 computer. I purchased my chip from Falk-Baker Associates, 382 Franklin Avenue, Nutley, New Jersey 07110, for \$14.95.

6560 Software

To produce colored characters, VIC addresses two blocks of memory at the same time. This address method produces 12 bits of data. The eight bit block of memory is called the Character Pointer Block (called the *screen memory* on the VIC-20).

The second block of memory is called the Character Color Block (called the *color nybble area* on the VIC-20). This block contains four-bit character color data.

VIC takes the character pointer data, left shifts it three times, and adds the result to the character cell base address contained in bits zero through three of register five.

VIC then puts the result on the address bus which addresses

another block of memory called the Character Cell Block (called character bit maps on the VIC-20). This block of memory is eight bits wide. The data obtained from this address is video information on a 8 x 8 character matrix. The matrix is eight bytes high and eight bits wide.

VIC takes the four-bit character color data and, if the Most Significant Byte is zero, the character matrix will be displayed in high resolution mode. If the MSB is one, the character matrix will be displayed in the multi-color mode.

When the high resolution mode is selected and when bit-3 of register-F is a zero, all one bits of the character cell data will be displayed in the background color and all zero bits will be in the foreground color. The three remaining bits of the character color data specify the color of the foreground. The color of the background is specified by bits four through seven of register-F. If bit-3 of register-F is 1, the one bits of the character cell data will be displayed in foreground color and the zero bits will be displayed in the background color.

If bit-3 of register-F is one, all the character cell matrix will have the same color background. If bit-3 is a zero, all the character cell matrix will have common character colors.

When the multicolor mode is selected (MSB of the character color data is one), there is a pairing of bits of the character cell data. The character matrix now is a 4 x 8 dot matrix with each dot's color determined by the code of each pair. The code has four possibilities: 00, 01, 10, 11. If a dot code is 00, its color is the background color specified by bits four through seven of register-F. If the code is 01, the dot color is the same as the external border color specified by bits zero through two of register-F. If the code is ten, the dot color is the foreground color specified by the three bits of the character color data. If the code is 11, the color of the dot is specified by bits four through seven of register-E.

VIC produces a TV raster of up to 22 columns by up to 23 rows of character matrix surrounded by a border. The base address of the character pointer block contains the first upper left pointer for that character matrix. The base address plus one of the character pointer blocks contains the pointer for the next right character matrix. It is the responsibility of the microprocessor unit to manipulate the pointers in the character pointer block of memory. A whole raster of repeated characters (character matrix) can be obtained by just repeating the pointers in the character pointer block.

6560 Hardware

The 6560 VIC has 14 address pins (A0-A13) and 12 data pins (D0-D11).

When the 02 clock is high, the microprocessor unit can place an address on the address pins and read or write data into any of the 16 eight-bit registers via data pin D0-D7. VIC decodes the address pins and selects registers zero through F when address 1000 through 100F (the VIC-20's VIC chip ignores the A 15 line) is placed on the address pins.

The address pins are input pins when 02 is one; if 02 is zero, then these pins are output address pins.

When the 02 clock is one, the microprocessor unit can also write or read data to the character pointer RAM, the character color RAM, and the character cell RAM. The character cell memory may be RAM, ROM, or both. The base address of the character cell block and/or the character pointer can be changed by modifying a register in the VIC.

When the 02 clock is low, the VIC addresses memory in such a way that the character pointer RAM and the character color RAM are selected at the same time. VIC must receive character pointer data on D0 through D7 and character color data on D8 through D11 at the same time.

The RAW pin four is an input only pin and must be driven by the microprocessor unit when 02 is one and held high when 01 is one.

Pins 38 and 39 are the master clock inputs. The 6560 VIC requires a 14.31818 MHz, two phase, five volt, non-overlapping signals. The master clock uses a standard 14.31818 MHz crystal (4x color) and the delay of 74LS gates to give a non-overlapping signal. Resistors R1 and R2 are used to give extra pull up to CMOS levels. CMOS gates don't seem fast enough for this clock.

Pins 35 and 36 are the system output clock used for system timing and driving the clock of the 6512 microprocessor unit (if used). A 6502 microprocessor unit can be used by feeding pin 36 to the 00 IN pin on the 6502. Removing the original 00 signal and wiring this new 00 is the only alteration needed on the mother microprocessor unit.

Because the 6502 address lines are active when 01 is 1, the expansion address lines to the VIC and its associated memory must be isolated from the microprocessor unit bus during this time.

The data bus should also be buffered and gated for this same reason.

The system clocks are five volt, non-overlapping, 1.02 MHz signals.

Pin 19 provides the sound output which must be fed to an

CHAPTER FIVE

amplifier to drive a speaker. The output impedance is approximately 1000 ohms.

Pin 3 is the output pin for the composite sync and the luminance signal. This pin is an open drain which makes it easy to shift to the needed voltage level for a RF modulator, TV first video amplifier, or a black and white CRT monitor.

In the following wiring diagram, diodes K3, K4, K5, and K6; resistors R5, R6, and R7; and C1 make up the level shifter for a TV output or video monitor output.

The VIC is a superior CRT controller for a B/W monitor due to the varying levels of luminance required for a color picture. VIC can produce varying shades of gray.

Pin 2 provides the composite color signal. This signal contains the color phase and amplitude information plus the 3.58 MHz burst signal. Pin 2 is a high impedance output buffer which can be applied to the first chroma amplifier of a TV, color monitor, or RF modulator.

Pins 17 and 18 are the input pins for the Pot-X and Pot-Y analog to digital converters. A pot is used to charge an external capacitor tied to the pot wiper and fed to pin 17 or 18. These pins are systematically pulled to ground after each charge voltage reading. The voltage is digitized and deposited in register 8 or 9.

Pin 37 is the light gun/pen pin. The voltage of triggering is approximately 2.5 volts on the falling edge. Holding this pin low clears registers 6 and 7. The values of registers 6 and 7 represent the horizontal and vertical positions of the current dot being scanned. The light gun/pen option is only available on a 6560-101, which is sometimes identified by a white dot on the case of the IC.

Address decoding must be provided so that the character color nybble RAM will be selected when the character pointer RAM is addressed by the VIC, but not selected when the microprocessor unit addresses the character pointer RAM. A data bus transceiver must be provided to isolate the nybble bus from the byte bus along with the logic for this transceiver. The logic must enable the microprocessor unit to read or write to the nybble bus when the 02 clock is one.

The expansion 02 may be used as the E(02) if the expansion 02 has no more than one LS gate delay and it is the true microprocessor unit 02. If these provisions can't be met, connect E(02) to the V02 (pin 36).

This author used addresses 1000 through 100F for the VIC; 2000 through 23FF for the character pointer block; 2400 through 27FF for

the character color block; and 2800 through 2BFF for the character cell block.

The addition of a 6560, a few RAM chips, and a few gates will free up and complement many a microprocessor unit.

Figure 1.

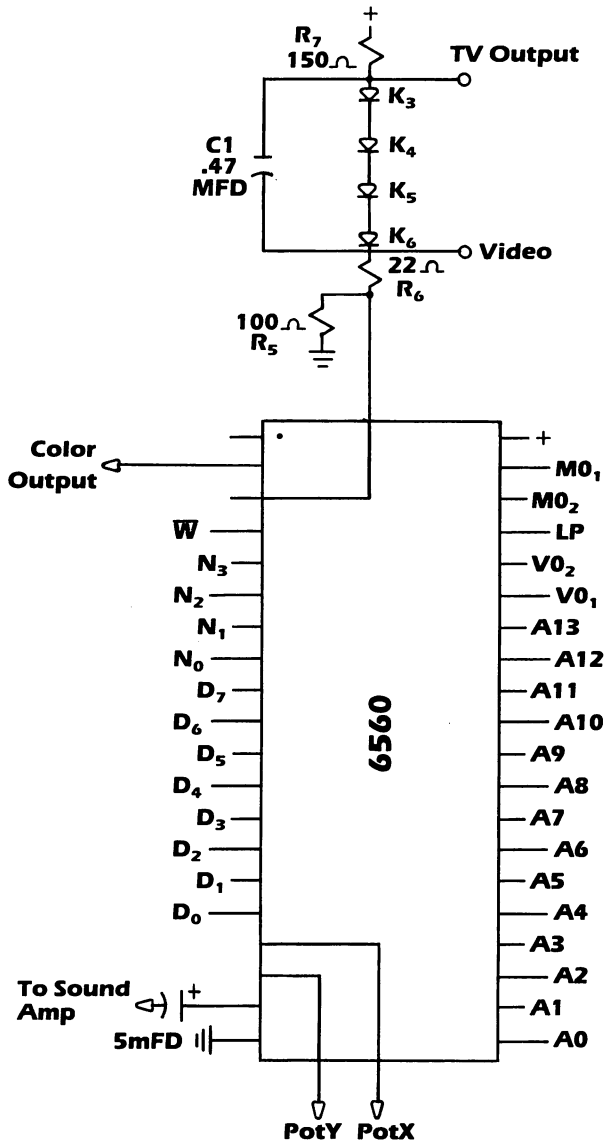


Figure 2. VIC Control Registers

Register	Data								Address
R0	1 = I 0 = N	Horiz. Center Vx4 Dots 6 5 4 3 2 1 0							1000
R1	Vertical Center Vx2 Dots 7 6 5 4 3 2 1 0								1001
R2	Cp A9	No. of Character Matrix Columns 6 5 4 3 2 1 0							1002
R3	RV 0	No. of Character Matrix Rows 5 4 3 2 1 0						1 = D 0 = S	1003
R4	Raster Value 8 7 6 5 4 3 2 1								1004
R5	Base Cp A13 A12 A11 A10				Base Char. Cell A13 A12 A11 A10				1005
R6	LP Horizontal Position 7 6 5 4 3 2 1 0								1006
R7	LP Vertical Position 7 6 5 4 3 2 1 0								1007
R8	Pot-X 7 6 5 4 3 2 1 0								1008
R9	Pot-Y 7 6 5 4 3 2 1 0								1009
RA	Sw 1 = On	Frequency Osc. -1 6 5 4 3 2 1 0							100A
RB	Sw 1 = On	Frequency Osc. -2 6 5 4 3 2 1 0							100B
RC	Sw 1 = On	Frequency Osc. -3 6 5 4 3 2 1 0							100C
RD	Sw 1 = On	Frequency Noise Gen. 6 5 4 3 2 1 0							100D
RE	Auxiliary Code 3 2 1 0				Audio Amplitude 3 2 1 0				100E
RF	Background Code 3 2 1 0				1 = B 0 = F		Border Code 2 1 0		100F

Abbreviations:

I = Interlace	R = Raster	B = Common Background
N = None Interlace	V = Value	F = Common Foreground
CP = Character Pointer	S = 8x8 Matrix	LP = Light Pen
Base Address	D = 16x8 Matrix	Sw = Switch

Color Code

Auxiliary/Background/Border/Foreground

0 BLACK	6 BLUE	C LIGHT MAGENTA
1 WHITE	7 YELLOW	D LIGHT GREEN
2 RED	8 ORANGE	E LIGHT BLUE
3 CYAN	9 LIGHT ORANGE	F LIGHT YELLOW
4 MAGENTA	A PINK	
5 GREEN	B LIGHT CYAN	

Browsing The VIC Chip

JIM BUTTERFIELD

An introduction, with experiments, to some of the dramatic effects you can achieve by changing the condition of your computer's VIC chip.

The computer is called VIC, for Video Interface Computer ... but there's a chip inside which is also called VIC, for Video Interface Chip. The chip bears the number 6560 or 6561; it's used to make good things happen on your television screen.

Beginners often don't realize that memory addresses are used for more than memory storage. In the VIC computer, addresses 36864 to 36879 may be PEEKed or POKEd. These locations are not used for memory – they hold controlling information for the VIC chip. We're going to look through those addresses, experimenting as we go. We may learn some new things about our computer.

● **Location 36864** (Hex 9000). Values 0 to 127 set the position of the left border on your screen. The usual value is five. Try the following quick "slide change" line:

```
FOR J = 5 TO 30:POKE 36864,J:NEXT J:FOR J = 30 TO 5  
STEP -1:POKE 36864,J:NEXT J
```

If you add 128 to the value in 36864, the screen will go into interlace mode. In most cases, all you'll notice if you POKE 36864,133 is a little "dither" in the screen detail. However, a few television sets are built in such a way that they won't work unless you set interlace mode with this POKE.

● **Location 36865** (Hex 9001). Values 0 to 255 set the position of the top border on your screen. The usual value is 25. Try making the screen "curtsey" with:

```
FOR J = 25 TO 45:POKE 36865,J:NEXT J:FOR J = 45 TO 25  
STEP -1:POKE 36865,J:NEXT J
```

● **Location 36866** (Hex 9002). Part of this location tells the chip how many columns on the screen. This will always be 22. But there's an extra – a value of 128 may be added to set "alternate

screen" mode. Normally, the 128 is added in, and you'll find 150 stored in this location. If you want to go to an alternate screen, remove the 128 element with POKE 36866,22 and the screen will now take its information from a new area. There are quite a few things you need to do if you wish to play with this – see "Alternate Screens" reprinted in this book.

• **Location 36867** (Hex 9003) is a busy one. In fact, it's always changing. Try typing ?PEEK(36867) several times and you'll see that you get different values – sometimes 46, sometimes 174. Let's ignore that extra 128 for the moment; we'll deal with it again when we describe the following location.

The basic value held in this location, normally 46, is the number of rows on the screen multiplied by two (23 rows, right?). You won't want to change this one.

There's one more thing hidden in this location, and it's important. If you add one to the value, the character generator will switch to "double character mode." This means that each character you type will occupy double the usual screen space.

This won't work automatically, however. If we want to draw characters that are twice as big, we must supply the VIC with "pictures" of the new characters; the old pictures won't do since they are not big enough to fill the new space. So prepare for a little confusion when you try this next experiment. Strange things will happen because we haven't built and connected up new character tables.

Type POKE 36867,47. The screen will go rather strange. Don't worry about it for the moment; just press the screen clear key (shifted, of course). The screen will clear, although the cursor looks rather odd. Not to worry, we'll forge right ahead.

The first character in the VIC's table of characters is the "@" symbol; the next is an "A," and then a "B" and so on. Now, type the @ key. Instead of getting the first character, we get the first two, one above the other. Try typing the "A" and you'll get B-over-C, the next two characters in the list.

What's happening here? Each character you type is filling double its normal screen area. In doing so, it's grabbing twice as much information from the "character picture" table... and, since that table hasn't been changed, that means two characters. Since the VIC knows (or at least thinks) that the character pictures are twice as big, it reaches further into that table for each character that it needs.

When you decide to use this feature, you'll write your own

character picture tables and everything will sort itself out. This feature is likely to be used most for high-resolution graphics. The elements of the character picture table will control individual dots, or pixels, on the screen.

You may bring your VIC back to normal by typing POKE 36867,46 but you'll need to type blind since the screen isn't much help. You might prefer to turn the computer off for a moment; when you turn it on again, everything will be back as it was.

• **Location 36868** (Hex 9004). This location changes continuously. It's connected with the high-bit (128 value) in the previous location. In principle, it tells you precisely where on the screen the picture is being drawn at this instant. In practice, it's not much use to BASIC programmers – by the time you read it, a different part of the screen will be active.

• **Location 36869** (Hex 9005). This is a very important address. It controls the location of two tables: the table where screen characters are held, and the table which holds the character pictures. Let's take them one at a time.

The screen table holds the 500 or so characters that are displayed on the screen. It's quite a job to calculate the screen address; let's take a shot at it.

Take the contents of location 36869, divide by 16, and throw away the remainder. That should give you a number from eight to 15. Subtract eight and double it, giving an even number from zero to 14. Now: if the contents of 36866 are 128 or greater, add one to get a value from zero to 15. Multiply the result by 512. At this point you should have a value from zero to 7680. That's where your screen table is located; it will normally be at location 7680, but it might move if you add extra memory.

That's quite a calculation; some of the things it implies deserve a separate article. For the moment, let's observe that the screen table address must always be in the range of zero to 7680, and must be a multiple of 512. If you wish to set up your own screen table within this range, do the calculation in reverse: divide by 512, subtracting one if odd, dividing by two, adding eight, and, finally, multiplying by 16. Whew! We can see that the "alternate screen" bit (128 value) in 36866 is really part of the much larger screen address.

The character picture table address is also defined in this location. We'd need to change this if we wanted to define our own characters, single or double. Of course, we'd also need to define character pictures for all characters we wished to print. The compu-

tation of the address is complex.

Take the contents of location 36869 and divide by 16. Now take the remainder – not the quotient – and, if it's greater than seven, subtract eight. On the other hand, if the remainder is not greater than seven, add 32. By this time, you'll have an adjusted remainder which is either less than seven or between 32 and 39. Multiply this value by 1024 and you've found your character picture table address. It will be in the range of either zero to 7168 or 32768 (the normal value) to 39936, and will be an exact multiple of 1024. If you wish to set up your own character picture table, you'll usually want it to point to a RAM address in the range of zero to 7168. In such a case, you'd reverse the calculation: take the address, divide by 1024 and add eight and you're there.

Don't forget that the screen table address and the character picture address are packed together into this location. You'll need to set them both at the same time. By the way, the official name for the two tables are the "Video Matrix" and the "Character Cell Table."

Feel free to play with this location; POKE values as you wish. But, unless you plan carefully, all you'll get is a crazy screen.

This was a tough one ... now we can try some easier locations.

● **Locations 36870 to 36871** (Hex 9006 and 9007). Here's your input from a light pen. No, a light pen isn't a ballpoint that weighs less than half an ounce – it isn't a pen at all. It's a device that looks like a pen, that plugs into your VIC. Point it at the screen, and these locations will tell you where you are pointing.

It is expected that Commodore will manufacture its own light pen soon. Many light pens have either a button or a spring-loaded switch in the tip which signals whenever the light pen operator wants attention. The switch is implemented in the VIC computer, but is not connected to the VIC chip (you'll find it mixed in with other things in location 37151).

You can read the X and Y positions of the light pen in locations 36870 and 36871 respectively. You won't read row and column values: the numbers will vary between zero and 255, and you'll need to do some calibration for the particular model of light pen that you have fitted.

Watch for "jitter" on these values. Even though the light pen doesn't move, the readings may jump about a little on successive readings. Depending on what you're doing, you may wish to use an averaging technique to make the readings smoother. Another method is called "hysteresis"; in simple terms, it means that a value is ignored unless it differs from previous readings by more than a

given threshold amount.

- **Locations 36872 to 36873** (Hex 9008 and 9009). These are paddle input values. Two paddles, such as Atari paddles, may be connected and their values read here. You may not be able to track the full range of rotation of the paddles.

Once again, watch for jitter on the input values here.

To keep the record straight, a joystick can also be connected to the VIC ... but the position of its button is not detected by the VIC chip. It arrives in other locations (37151 and 37152).

- **Locations 36874 to 36876** (Hex 900A to 900C). These are VIC's voices. Setting a value of 128 or higher into any of these locations produces sound; the value you POKE produces the pitch. By POKEing two or three locations, you can produce harmony. All voices are controlled by the sound level which is set at address 36878; try POKE 36878,15 before you play with the voices so that you'll get good volume. A value of less than 128 in any of the voice locations makes that voice silent.

It's interesting to note that the voice controlled by 36874 is the softest, and the voice at 36876 is the sharpest. So you'd use 36876 to carry the melody, and the two other voices as the sidemen.

- **Location 36877** (Hex 900D). This is similar to the music voices, except that it generates noise. A value of 128 or more produces noise. The higher the value, the higher the pitch of the noise (from growl to hiss). Once again, this is controlled by the sound level of 36878.

- **Location 36878** (Hex 900E). If the number in here is less than 16, it represents the sound amplitude (see the four previous locations). If it's 16 or more, an extra factor is at work: multi-color.

Normally, each character position on the VIC contains only two colors: background and foreground. If we decide to use multi-color, we can add an extra two colors to each character: the border color plus one more that we may select. We select this color in the high part of location 36878. If you divide the contents of this location by 16, discarding the remainder, you'll get the designation of the "auxiliary color."

Interestingly, each character on VIC's screen is independently selected as two-color or multi-color, allowing us to have a mixed screen. This is done in the color nybble table which sets each character's color. Try the following: Clear the screen and type the letter A in the upper left-hand corner. Now go to a new line and type POKE 38400,8. You'll see that the letter A has suddenly turned

weird and multi-colored, but the rest of the screen is unchanged. Notice that we did not POKE the VIC chip, but an entirely different memory location that is keyed to the one screen address. To do the job properly, you'll need to define your own character pictures.

• **Location 36879** (Hex 900F). The last location in the VIC chip, but a busy one. Let's break it down into its three elements.

Divide the contents of this location by 16, and note the result as "Screen Background Color." Now take the remainder; if it's eight or more, subtract and note: Foreground/Background = ON. The remaining value of zero to seven can be labelled "Frame Color."

The Frame Color is a favorite of mine; it's an easy signal to the user of some situation I want to tell him about without affecting the contents of the screen itself. If there's a danger, an error, or a game explosion, I can flip the border to red with POKE 36879,26 and later restore it with POKE 36879,27. Another example: rather than typing a PLEASE WAIT message, I might walk the border through a range of colors so that the user can tell something is happening.

Screen Background color can be a very nice psychological support. If you set up a system so that accounts receivable can be done on one background color and accounts payable on a different one, the user can be "keyed" to recognize that he's in the right program. It's a little like decorating each floor of a building in a different color so that people won't get the wrong one. Try combinations such as POKE 36879,155 and see how you like the effect.

Now for the Foreground/Background business. Normally (F/B = On) we know that we can type characters of many colors on a single color background. Sometimes, it can be very handy to do the opposite; in other words, we want to type single color characters on a background whose color may vary from character to character.

Try POKE 36879,19. Now clear the screen and type a few characters. Change the color and type some more. Do you see what's happening? You are changing the color of the background, not the color of the characters themselves.

By playing around with these locations, you can discover potential that you never knew existed. Once you know it's there, you can then exploit it for your own special effects.

There's a rich variety of controls and information available in the VIC chip. You may not need to use them all – but isn't it fascinating to play around?

VIC 6560 Chip

\$9000	Inter-lace	Left Margin (= 5)		36864
\$9001	Top Margin (= 25)			36865
\$9002	Scrn Ad bit 9	# Columns (= 22)		36866
\$9003	bit 0	# Rows (= 23)	Double Char	36867
\$9004	Input Raster Value: bits 8-1			36868
\$9005	Screen Address bits 13-10		Character Address bits 13-10	36869
\$9006	Horizontal			36870
\$9007	Vertical			36871
\$9008	X			36872
\$9009	Y			36873
\$900A	ON	Voice 1		36874
\$900B	ON	Voice 2		36875
\$900C	ON	Voice 3		36876
\$900D	ON	Noise		36877
\$900E	Multi-Color Mode (= 0)		Sound Amplitude	36878
\$900F	Screen Background Color		Foregnd /Backg	36879
			Frame Color	

VIC Memory – The Uncharted Adventure

DAVID BARRON / MICHAEL KLEINERT

All computer enthusiasts have at one time or another wondered about all the unexposed memory locations of their computers. These can range from locations which disable the RUN/STOP key to disabling the entire keyboard. Here's a short list of useful memory locations.

Location: 650 Normal Value: 0

As you already know, the two cursor control keys, the INSERT/DELETE key, and the SPACE BAR automatically repeat. By POKEing different values into location 650, you can change this function.

POKE 650,100: Disables repeat of all keys.

POKE 650,255: Enables repeat of all keys.

POKE 650,0: Restores keyboard to normal operation.

Location: 808 Normal Value: 112

This is one of the most confusing locations we stumbled across. Many values POKEd into this location may cause an adverse effect in the computer's operation and may result in a serious crash.

POKE 808,100: Disables RUN/STOP and RESTORE keys.

POKE 808,112: Returns to normal.

Warning: POKEing 808,100 also changes the appearance of the program listing, but the program will still run normally.

Location: 649 Normal Value: 10

This location controls the number of characters that may be stored in the keyboard buffer. For example, POKEing 649,5 will allow only five characters to be stored in the buffer.

POKE 649,0: Totally disables keyboard.

POKE 649,10: Restores keyboard to normal.

Whenever characters are typed into the computer, they first must pass through the keyboard buffer before they register. This buffer will normally store up to ten characters. If it is set to zero, the buffer will store zero characters. If no characters are stored in the buffer none will pass into the computer.

Location: 198 Normal Value: 0-10

This location determines the number of characters presently in the keyboard buffer. If it is set to zero, the keyboard buffer will be emptied. Using zero before a GET statement will keep unwanted information from being entered.

POKE 198,0: Clears keyboard buffer.

No need to restore to normal.

Location: 36881 Normal Value: 24

This location can be used to position the entire screen workspace. The values to be POKEd range from 0 to 255. Increasing or decreasing the current value by one causes a very small amount of movement. This can be used to create very smooth animation. A very nice effect is to scroll up a display seemingly from nowhere. A short example of this is demonstrated in Program 1.

POKE 36881,X (where X is greater than 24):

Moves entire screen down.

POKE 36881,X (where X is less than 24):

Moves entire screen up.

POKE 36881,24: Resets screen to normal position.

Location: 36883 Normal Value: 46

In addition to being able to position the entire screen vertically, you may also control how many lines of text are visible. It is controlled by POKeing even-numbered values from 0 to 46. If an odd number is POKEd, the computer will react strangely, but can easily be reset by using the RUN/STOP and RESTORE keys. Unlike location 36881, location 36883 works line by line. Every time location 36883 is increased or decreased by two, the amount of visible screen workspace changes by one line.

POKE 36883,X (where X is less than 46):

Decreases amount of visible screen workspace.

POKE 36883,46: Restores screen to normal.

CHAPTER FIVE

```
10 A=36881:POKE A-3,15
20 POKE A,128: REM REPOSITIONS WORKSPACE SO I
  T IS NOT VISIBLE
30 PRINT"{CLEAR}{10 DOWN}{05 RIGHT}{GRN}DISPL
  AY DEMO"
40 PRINT"{05 RIGHT}{RED}EEEEEEEEEEEEEE
50 FOR T=128 TO 24 STEP-1: REM LOOP TO MOVE T
  HE SCREEN UP
55 POKE A,T:POKE A-5,256-T
60 FOR P=1 TO 10:NEXT P:NEXT : REM DELAY TO S
  LOW MOVEMENT
65 FOR T=1 TO 600:NEXT:REM  WAITS SEVERAL SEC
  ONDS
70 FOR T=46 TO 0 STEP-2:REM LOOP TO REDUCE VI
  SIBLE WORKSPACE
75 POKEA+2,T:POKEA-5,128+2*T:FORP=1TO40:NEXTP,T
80 PRINT "{CLEAR}{06 DOWN}{09 RIGHT}";CHR$(14
  4);"B{DOWN}Y{02 DOWN}"
90 PRINT TAB(3);"{REV}MICHAEL KLEINERT
100 PRINTTAB(9);"{DOWN}{GRN}AND
110 PRINTTAB(5);"{DOWN}{BLU}DAVID BARRON
120 FOR T=0 TO 46 STEP 2:REM INCREASES AMOUNT ~
  OF VISIBLE WORKSPACE
125 POKE A+2,T:POKE A-5,128+2*T:FOR P=1 TO 40:
  NEXT P,T
130 FOR P=1 TO 700:NEXT
140 POKE A-5,0:POKE A-3,0
150 FOR P=1 TO 5000:NEXT:PRINT"{BLU}{CLEAR}
999 END
```

Memory Map Above Page Zero

JIM BUTTERFIELD

Memory Map

0100-103E	256-318	Tape error log
0100-01FF	256-511	Processor stack area
0200-0258	512-600	BASIC input buffer
0259-0262	601-610	Logical file table
0263-026C	611-620	Device # table
026D-0276	621-630	Sec Adds table
0277-0280	631-640	Keybd buffer
0285	645	Serial bus timeout flag
0286	646	Current color code
0287	647	Color under cursor
0288	648	Screen memory page
0289	649	Max size of keybd buffer
028A	650	Repeat all keys
028B	651	Repeat speed counter
028C	652	Repeat delay counter
028D	653	Keyboard Shift/Control flag
028E	654	Last shift pattern
028F-0290	655-656	Keyboard table setup pointer
0291	657	Keymode (Kattacanna)
0292	658	0=scroll enable
0293	659	VIC chip control
0294	660	VIC chip command
0295-0296	661-662	Bit timing
0297	663	RS-232 status
0298	664	# bits to send
0299-029A	665	RS-232 speed/code
029B	667	RS232 receive pointer
029C	668	RS232 input pointer
029D	669	RS232 transmit pointer
029E	670	RS232 output pointer
029F-02A0	671-672	IRQ save during tape I/O
0300-0301	768-769	Error message link
0302-0303	770-771	BASIC warm start link
0304-0305	772-773	Crunch BASIC tokens link
0306-0307	774-775	Print tokens link
0308-0309	776-777	Start new BASIC code link
030A-030B	778-779	Get arithmetic element link
0314-0315	788-789	Hardware interrupt vector (EABF)
0316-0317	790-791	Break interrupt vector (FED2)

CHAPTER FIVE

0318-0319	792-793	NMI interrupt vector	(FEAD)
031A-031B	794-795	OPEN vector	(F40A)
031C-031D	796-797	CLOSE vector	(F34A)
031E-031F	798-799	Set-input vector	(F2C7)
0320-0321	800-801	Set-output vector	(F309)
0322-0323	802-803	Restore I/O vector	(F3F3)
0324-0325	804-805	INPUT vector	(F20E)
0326-0327	806-807	Output vector	(F27A)
0328-0329	808-809	Test-STOP vector	(F770)
032A-032B	810-811	GET vector	(F1F5)
032C-032D	812-813	Abort I/O vector	(F3EF)
032E-032F	814-815	USR vector	(FED2)
0330-0331	816-817	LOAD link	
0332-0333	818-819	SAVE link	
033C-03FB	828-1019	Cassette buffer	
0400-0FFF	1024-4095	3K RAM expansion area	
1000-1FFF	4096-8191	Normal BASIC memory	
2000-7FFF	8192-32767	Memory expansion area	
8000-8FFF	32768-36863	Character bit maps	
9000-900F	36864-36879	Video Interface Chip	
9110-912F	37136-37167	6522 Interface Chips	
9400-95FF	37888-38399	Alternate Colour Nybble area	
9600-97FF	38400-38911	Main Colour Nybble area	
A000-BFFF	40960-49151	Plug-in ROM area	
C000-FFFF	49152-65535	ROM: BASIC and Operating System	

VIC Usage: The 6522-A

\$9110	RS-232 or Parallel User Port						37136
\$9111	unused — see \$911F						37137
\$9112	DDRB (for \$9110)						37138
\$9113	DDRA (for \$911F)						37139
\$9114	T1-L RS-232 Send speed;						37140
\$9115	T1-H Tape write timing						37141
\$9116	T1 Latch						37142
\$9117	T1 Latch						37143
\$9118	T2-L RS-232						37144
\$9119	T2-H Input timing						37145
\$911A	Shift Register (not used)						37146
\$911B	T1 Control	T2C	SR Control		PBLE	PALE	37147
\$911C	RS-232 Send		Cb1 cont	Tape motor		CA1 cont	37148
\$911D	NMI:	T1	T2	CB1: RS-232 IN	CA1: RSTR butn	37149
\$911E							37150
\$911F	ATN out	Tape sens	Joystick Butn 0 1 2			In; Serl	In; Clok 37151

VIC Usage: The 6522-B

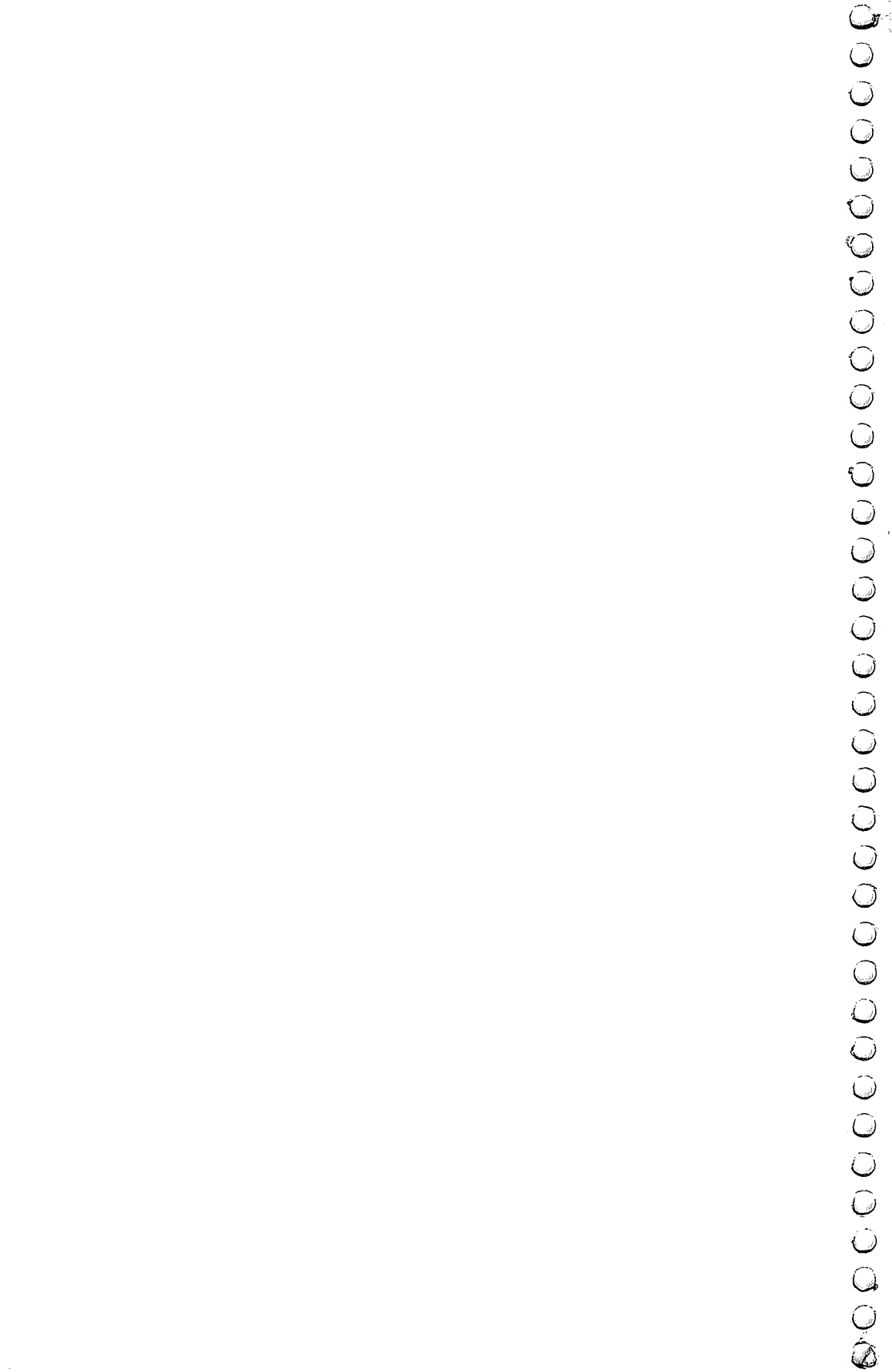
\$9120	RS232 in Joy 3	Tape out	37152
	---- Keyboard Row Select ----		
\$9121	Keyboard Row Input		37153
\$9122	DDRB (for \$9120)		37154
\$9123	DDRA (for \$9121)		37155
\$9124	T1-L	Cassette tape read;	37156
\$9125	T1-H	Keyboard (interrupt)	37157
\$9126	T1 Latch	timing	37158
\$9127	T1 Latch		37159
\$9128	T2-L	Serial Bus timeout;	37160
\$9129	T2-H	Cassette R/W timing	37161
\$912A	Shift Register (unused)		37162
\$912B	T1 Control T2C	SR Control PBLE PALE	37163
\$912C	Serial bus out	Clock line out CA1 Contl	37164
\$912D	IRQ: T1 T2	CA1: Tape	37165
\$912E			37166
\$912F	unused — see \$9121		37167

VIC Usage: The 6560 Video Interface Chip

\$9000	Inter-lace (0)	Left Margin (=5)	36864
\$9001	Top Margin (=25)		36865
\$9002	Sc. Adds b9	# Columns (=22)	36866
\$9003	b0	# Rows (=23)	36867
		Double Char	
\$9004	Raster Value In: 68-b1		36868
\$9005	Screen Address b13-b10	Character Address b13-b10	36869
\$9006	Light Pen	Horizontal	36870
\$9007	(option)	Vertical	36872
\$9008	Potentiometer	X	36872
\$9009	Sense (option)	Y	36874
\$900A	△	Voice 1	36874
\$900B	△	Voice 2	36875
\$900C	△	Voice 3	36876
\$900D	△	Noise	36877
\$900E	Multi-Colour Mode (=0)	Sound Amplitude	36878
\$900F	Screen Background	Fore/Back-Ground	36879
		Frame Colour	

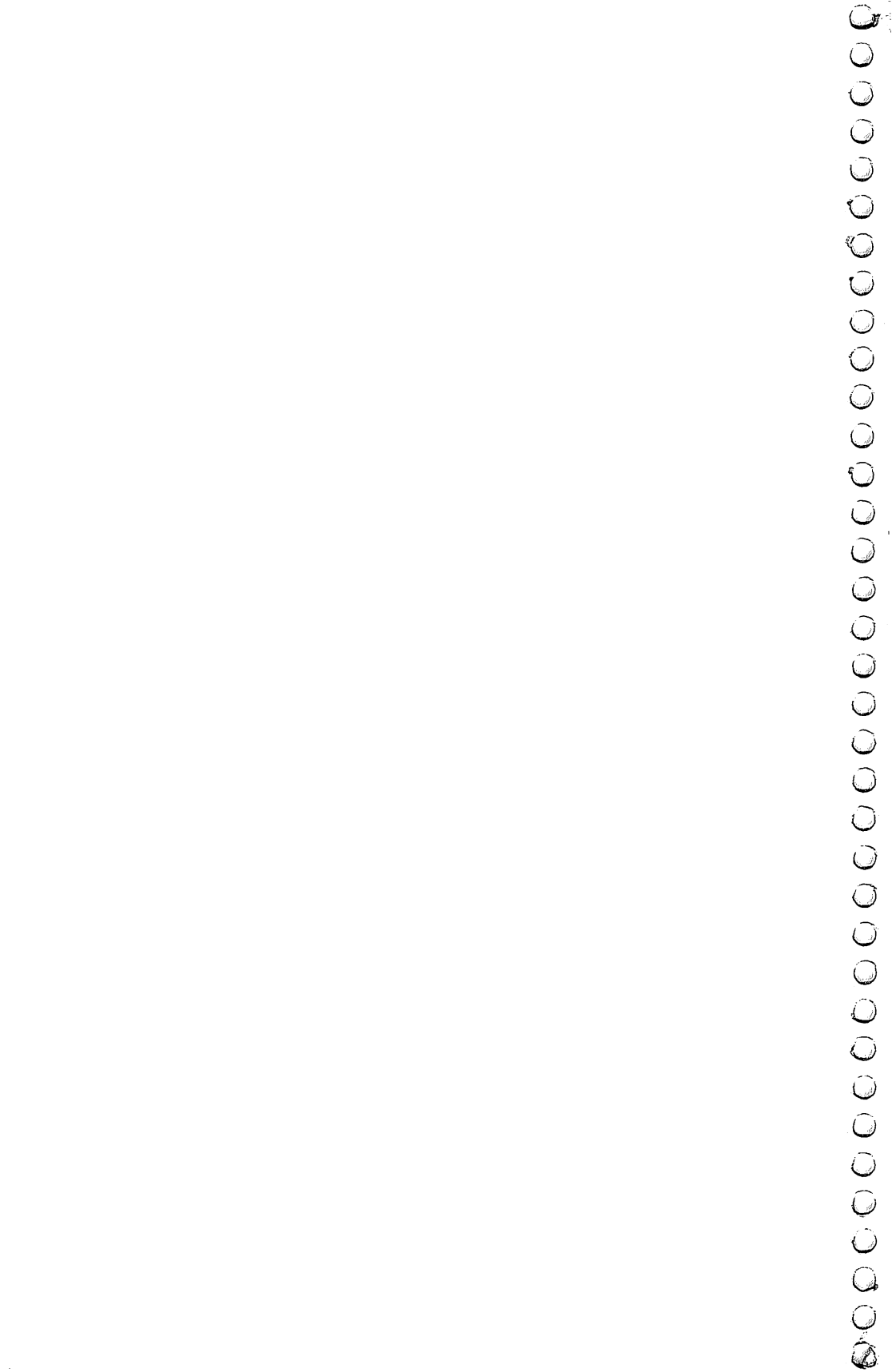
Values, where shown, are the normal default VIC values.

Light Pen and Potentiometer are implemented in hardware but not used in ROM programs.



CHAPTER SIX

MACHINE LANGUAGE



TINYMON1: A Simple Monitor For The VIC

JIM BUTTERFIELD

TINYMON1 is a full-featured machine language monitor. It lets you edit, enter, run, save, and load machine language programs – all this in only 760 bytes of memory.

One of the things you may miss on the VIC is a machine language monitor; it's not there.

Commodore will be releasing a very powerful monitor on a plug-in cartridge, and serious programmers will certainly want to use it. But for occasional use, a tape-loadable monitor might be very handy.

Here's an early version that may be of use. It should fit on any VIC, with or without extra memory added; and it honors all the commands from the built-in monitors we know from PET/CBM usage. One minor syntax change: the two addresses of the Memory display command (.M) should be separated by a space rather than a comma.

It's not really practical to type TINYMON1 directly into a VIC. DATA statements in decimal would take up more room than is available in small VICs; and hex entry would need a monitor to be in place already. So I've prepared the program so that it can be entered on a PET and saved on tape. After it's been created once, the VIC can make its own copies. You'll need a PET with Upgrade ROM or 4.0 ROM to do the job, since the Original ROM PETs don't have a machine language monitor and things would get too complicated (another article reprinted in this book shows how to enter TINYMON1 directly into a VIC if you lack access to a PET).

TINYMON1 loads like a BASIC program, and copies can be made with a simple LOAD and SAVE sequence as you would do with BASIC. When you load TINYMON1 and say RUN, however, some interesting things happen ... the monitor system is repacked

into the top of memory, and it will stay there until you turn the power off. You can say .X to return to BASIC and load and run BASIC programs, providing they are not too big. TINYMON1 grabs about 760 bytes of memory, so you lose a little space.

Find A Zero

Once you're back in BASIC, the question arises: how can you invoke TINYMON1 when desired? Not an easy trick, since memory is more mobile in the VIC than in the PET/CBM. The thing to do is to find a zero value in memory and SYS to that location. If you have a basic (5K) VIC, SYS 4096 is safe. The sure way is to PEEK first and ensure that there's a zero there (location 10 is often zero).

TINYMON1 must be considered preliminary. It was designed with two major considerations: to use minimum space, and to automatically load into any VIC regardless of the memory fitted. The space consideration is fairly obvious: with 3500-odd bytes available on a small VIC, you want to use up as little as possible. The automatic load feature was tricky to implement; VIC may relocate programs as it loads. What's more, the screen area tends to move around as you add memory.

I scratched my head over the .S (Save) command. If VIC automatically relocates programs during loading, will a SAVEd machine language program be safe? As it turns out, VIC has a new tape format available – when a tape is written, it may be defined as “absolute” and will not relocate when it loads. This seems the best compromise, but it has one drawback – the PET/CBM won't load this type of tape. Perhaps that's a design decision that will need to be revised...

Finding Space In Zero Page

VIC is desperately short of zero page space; machine language programmers will have to cope with the shortage as best they can. I have used the same locations that the big Commodore monitor is expected to use. There's a difference, however: the Commodore job will swap out selected parts of zero page and put them back later; I didn't want to give up the space for that kind of luxury. As a result, you may be annoyed by some locations that are disturbed by TINYMON1.

For those unfamiliar with the PET/CBM machine language monitor, the commands are:

.R – display 6502 registers;

Users can use screen editing to type over a display and change

the registers;

.M FFFF TTTT – display memory (from ... to);

Users can use screen editing to type over a display and change memory;

.X – exit to BASIC;

It may be wise to type CLR in BASIC after exiting;

.G AAAA – GOTO (execute) address;

.S "PPPP",01,FFFF,TTTT – Save (program-name, device, from, to);

.L "PPPP" – Load (program-name)

There's a delicate tradeoff between features and memory space. There will undoubtedly be other small monitors with a different balance. In any case, I wrote one because I had nothing ... and others in the same position will undoubtedly greet TINYMON1 with glad cries.

Program 1. TINYMON1.

Enter on a PET/CBM, using the machine language monitor. Do not try to RUN, but follow your entry with the checksum program, Program 2.

First, make the following change:

```
.. 0028 01 04 14 08 14 08 14 08
```

Now, enter TINYMON1:

```
.: 0400 00 18 04 64 00 99 22 93
.: 0408 11 11 12 1D 1D 1D 20 54
.: 0410 49 4E 59 4D 4F 4E 20 00
.: 0418 31 04 6E 00 99 22 11 20
.: 0420 4A 49 4D 20 42 55 54 54
.: 0428 45 52 46 49 45 4C 44 22
.: 0430 00 4C 04 78 00 9E 28 C2
.: 0438 28 34 33 29 AA 32 35 36
.: 0440 AC C2 28 34 34 29 AA 30
.: 0448 37 38 29 00 00 00 EA EA
.: 0450 A5 2D 85 22 A5 2E 85 23
.: 0458 A5 37 85 24 A5 38 85 25
.: 0460 A0 00 A5 22 D0 02 C6 23
```

CHAPTER SIX

```

.: 0468 C6 22 B1 22 D0 3C A5 22
.: 0470 D0 02 C6 23 C6 22 B1 22
.: 0478 F0 21 85 26 A5 22 D0 02
.: 0480 C6 23 C6 22 B1 22 18 65
.: 0488 24 AA A5 26 65 25 48 A5
.: 0490 37 D0 02 C6 38 C6 37 68
.: 0498 91 37 8A 48 A5 37 D0 02
.: 04A0 C6 38 C6 37 68 91 37 18
.: 04A8 90 B6 C9 BF D0 ED A5 37
.: 04B0 85 33 A5 38 85 34 6C 37
.: 04B8 00 00 00 00 00 00 00 00
.: 04C0 BF 78 AD FE FF 00 AE FF
.: 04C8 FF 00 8D 16 03 8E 17 03
.: 04D0 A9 80 20 90 FF 58 00 00
.: 04D8 68 85 05 68 85 04 68 85
.: 04E0 03 68 85 02 68 85 01 68
.: 04E8 85 00 00 BA 86 06 38 A5
.: 04F0 01 E9 02 85 01 A5 00 00
.: 04F8 E9 00 00 85 00 00 20 B2
.: 0500 FE 00 A2 42 A9 2A 20 DB
.: 0508 FD 00 A9 52 D0 1C A9 3F
.: 0510 20 D2 FF 20 B2 FE 00 A9
.: 0518 2E 20 D2 FF A9 00 00 85
.: 0520 27 20 40 FE 00 C9 2E F0
.: 0528 F9 C9 20 F0 F5 A2 07 DD
.: 0530 E6 FF 00 D0 12 85 1C 8A
.: 0538 0A AA BD EE FF 00 85 C1
.: 0540 BD EF FF 00 85 C2 6C C1
.: 0548 00 00 CA 10 E6 4C 4B FD
.: 0550 00 20 BD FD 00 90 F8 20
.: 0558 EE FD 00 20 BD FD 00 90
.: 0560 F0 20 EE FD 00 20 4C FE
.: 0568 00 F0 1F 20 B2 FE 00 A2
.: 0570 2E A9 3A 20 DB FD 00 20
.: 0578 C5 FD 00 A9 05 20 6F FE
.: 0580 00 A5 C3 C5 C1 A5 C4 E5
.: 0588 C2 B0 DF 4C 50 FD 00 4C
.: 0590 50 FD 00 20 FE FD 00 85
.: 0598 C1 86 C2 60 A5 C2 20 CC
.: 05A0 FD 00 A5 C1 48 4A 4A 4A
.: 05A8 4A 20 E4 FD 00 AA 68 29
.: 05B0 0F 20 E4 FD 00 48 8A 20

```

```
..: 05B8 D2 FF 68 4C D2 FF 18 69
..: 05C0 F6 90 02 69 06 69 3A 60
..: 05C8 A2 02 B5 C0 48 B5 C2 95
..: 05D0 C0 68 95 C2 CA D0 F3 60
..: 05D8 20 0D FE 00 90 07 AA 20
..: 05E0 0D FE 00 90 01 60 4C 4B
..: 05E8 FD 00 A9 00 00 85 2A 20
..: 05F0 40 FE 00 C9 20 F0 F9 20
..: 05F8 20 FE 00 90 17 20 40 FE
..: 0600 00 C9 30 90 10 20 35 FE
..: 0608 00 06 2A 06 2A 06 2A 06
..: 0610 2A 05 2A 85 2A 38 60 C9
..: 0618 3A 08 29 0F 28 90 02 69
..: 0620 08 60 20 CF FF C9 0D D0
..: 0628 F8 68 68 4C 50 FD 00 A5
..: 0630 91 C9 FE D0 05 08 20 CC
..: 0638 FF 28 60 20 61 FE 00 2C
..: 0640 2D 91 30 F8 60 20 4C FE
..: 0648 00 D0 08 A9 03 85 9A A9
..: 0650 00 00 85 99 60 85 1E A0
..: 0658 00 00 20 AF FE 00 B1 C1
..: 0660 20 CC FD 00 20 A4 FE 00
..: 0668 C6 1E D0 F1 60 20 0D FE
..: 0670 00 90 0B A2 00 00 81 C1
..: 0678 C1 C1 F0 03 4C 4B FD 00
..: 0680 20 A4 FE 00 C6 1E 60 A9
..: 0688 02 85 C1 A9 00 00 85 C2
..: 0690 A9 05 60 E6 C1 D0 06 E6
..: 0698 C2 D0 02 E6 27 60 A9 20
..: 06A0 2C A9 0D 4C D2 FF A2 00
..: 06A8 00 BD D0 FF 00 20 D2 FF
..: 06B0 E8 E0 16 D0 F5 20 B2 FE
..: 06B8 00 A2 2E A9 3B 20 DB FD
..: 06C0 00 A5 00 00 20 CC FD 00
..: 06C8 A5 01 20 CC FD 00 20 99
..: 06D0 FE 00 20 6F FE 00 4C 50
..: 06D8 FD 00 20 FE FD 00 85 01
..: 06E0 86 00 00 20 99 FE 00 85
..: 06E8 1E 20 83 FE 00 D0 FB F0
..: 06F0 EA 20 BD FD 00 A9 05 85
..: 06F8 1E 20 83 FE 00 D0 FB F0
..: 0700 DC 20 CF FF C9 0D F0 07
```


CHAPTER SIX

```
..: 0708 20 BD FD 00 85 01 86 00
..: 0710 00 A6 06 9A A5 00 00 48
..: 0718 A5 01 48 A5 02 48 A5 03
..: 0720 A6 04 A4 05 40 78 A6 06
..: 0728 9A 6C 02 C0 4C 4B FD 00
..: 0730 A0 01 84 BA 84 B9 88 84
..: 0738 B7 84 90 84 93 A9 02 85
..: 0740 BC A9 40 85 BB 20 CF FF
..: 0748 C9 20 F0 F9 C9 0D F0 1A
..: 0750 C9 22 D0 D9 20 CF FF C9
..: 0758 22 F0 26 C9 0D F0 0B 91
..: 0760 BB E6 B7 C8 C0 10 F0 C5
..: 0768 D0 EA A5 1C C9 4C D0 E2
..: 0770 A9 00 00 20 D5 FF 20 58
..: 0778 FE 00 A5 90 29 10 D0 F0
..: 0780 4C 50 FD 00 20 CF FF C9
..: 0788 0D F0 E2 C9 2C D0 F0 20
..: 0790 0D FE 00 29 0F F0 D3 C9
..: 0798 03 F0 FA 85 BA 20 CF FF
..: 07A0 C9 0D F0 CA C9 2C D0 E6
..: 07A8 20 BD FD 00 20 CF FF C9
..: 07B0 2C D0 F4 20 FE FD 00 85
..: 07B8 AE 86 AF 20 CF FF C9 20
..: 07C0 F0 F9 C9 0D D0 EC A5 1C
..: 07C8 C9 53 D0 F8 20 B2 FE 00
..: 07D0 A9 01 85 B9 20 82 F6 4C
..: 07D8 50 FD 00 0D 20 20 50
..: 07E0 43 20 20 53 52 20 41 43
..: 07E8 20 58 52 20 59 52 20 53
..: 07F0 50 4D 52 58 47 3A 3B 4C
..: 07F8 53 86 FD 00 B7 FE 00 23
..: 0800 FF 00 02 FF 00 F4 FE 00
..: 0808 E1 FE 00 2D FF 00 2D FF
..: 0810 00 1B FD 00 00 00 00
```

Whew! TINYMON1 for the VIC is now entered. Check it with the following program:

Program 2. A Checking Program.

Type the following direct line on the screen of your PET/CBM:

**forj = 1024to2071step8:t = 0:fork = jtoj + 7:t = t + peek
(k):next:?t;next**

You should see the following numbers appear on the screen of your PET. Check them carefully. Each one represents one line of entry, starting at 0400 hexadecimal. If any of these totals are wrong, you've entered the line incorrectly.

The numbers in parentheses appearing to the right won't appear on your screen; they are there to help you locate an incorrect line.

When you are satisfied that the program is entered correctly, SAVE it to cassette tape. It may now be loaded into your VIC.

462 255 506 399 575 541 592 511	(0400)
769 620 756 780 802 910 886 853	
801 784 876 840 835 1383 753 0	
1422 589 816 720 584 680 535 576	
944 972 1130 845 876 1357 1010 1188	(0500)
1311 852 898 1109 1125 897 809 1021	
1340 1078 1005 1212 905 902 770 1239	
762 1133 1388 652 659 629 1072 803	
748 150 617 413 1020 1030 1057 818	(0600)
944 844 705 831 939 1072 639 1033	
943 824 1137 970 929 1149 1395 940	
654 840 807 926 706 1146 1015 1146	
1175 742 563 645 695 860 1064 1042	(0700)
1235 1202 1355 922 1445 1346 789 1068	
1104 1204 975 1306 1339 1169 1168 1210	
1340 1204 972 522 460 520 591 942	
1010 1079 280	(0800)

Entering TINYMON1 Directly Into Your VIC-20

RUSSELL KAVANAGH

If you are interested in learning or working with machine language, a "monitor" program is necessary to show you the condition of your machine's "registers," to allow you to easily enter hexadecimal numbers, and to permit direct SAVEing and LOADING of portions of memory. Elsewhere in this book, you'll find Jim Butterfield's "TINYMON1," a monitor for VIC. Because of memory constraints, it requires a PET to help with entering the program. This article explains how to type it into your VIC if a PET computer is not available to you.

When I first saw Jim Butterfield's article "TINYMON1: A Simple Monitor For The VIC" (**COMPUTE!**, January 1982, #20, pg. 176), I was very interested, since I was in need of a simple way to enter small machine language programs without the laborious technique of using BASIC POKE statements. But the fourth paragraph quickly diffused my elation, since it stated I needed a PET to enter the program.

I don't know any PETs personally, so it at first looked as if I wouldn't be able to take advantage of TINYMON1. But a dedicated hobbyist, armed with motivation and a VIC memory map, can be unstoppable. I'm going to show you how to enter TINYMON1 directly into the VIC.

As I saw it, there were three problems to solve. To begin with, where in the VIC's memory is the program supposed to run? Second, how can the program be entered? Last, how can it be saved on cassette? Not knowing exactly where to start, I decided to take a closer look at the TINYMON1 listing and see what I could learn.

Where Does It Load?

Since I had no idea where TINYMON1 was supposed to run in the

VIC memory, I tried disassembling the listing. (Disassembly, by the way, is a process of taking the machine language op code and converting it to the mnemonic, or "name," of the command.) Disassembly, I theorized, would perhaps lead to some clues to the program's location in memory. But, several bytes into the program, I realized that something was fishy. Some of the codes weren't even legitimate 6502 commands! In fact, many were ASCII characters. The first of the program, then, was apparently data, not program codes. By now, though, I was tiring of this hand disassembly process. I decided to turn to the VIC memory map for help.

According to the memory map, program memory for a VIC with no more than 8K begins at 1000 hex (4096 decimal). It seemed logical that any program loaded from cassette or entered from the keyboard would begin there. But the TINYMON1 program began at 0400 hex, not 1000 hex. This, I figured, could simply be the difference in the VIC and PET memory configurations. On a hunch, I tried POKEing in the first several bytes from the listing, starting at 4096 decimal. When I tried the BASIC LIST command, I got part of a BASIC program statement!

So, the first part of the TINYMON1 program was actually a BASIC program. A little more experimentation developed the syntax, or construction, of a BASIC statement, shown in Figure 1. I won't try to go into detail here on the explanation of the syntax, but if you are interested, more information is in the *VIC Programmer's Guide*, now available from Commodore dealers.

The important item to notice is that the first two bytes in each statement point to the beginning of the next statement. Since I intended to enter the code starting at 1000 hex rather than 0400 hex, I knew I'd need to alter those pointers. For example, the first pointer (seen in the second and third bytes of the listing) was 0418 hex (the low order byte is always first), which would change to 1018 hex. Note that the first byte of any VIC BASIC program is apparently 0, which Butterfield refers to in his article when he describes how to re-enter TINYMON1. At this point, I hoped that I had solved the problem of *where* to load TINYMON1. The next problem to be solved was *how* to load it.

How Can It Be Loaded?

I could see several possibilities for entering the program. The most common way to load a machine language program is using a BASIC program with the machine language coded in DATA statements, and POKEing them into memory. [*This is called a BASIC loader.*] But

Butterfield said this wouldn't work, since the minimal VIC with only 3.5K of memory doesn't have enough room to contain the monitor and the BASIC program needed to enter it. A little arithmetic confirms that constraint.

Since the POKE function requires decimal numbers, the DATA statements representing the machine code would most logically be in decimal. Most, then, will require three bytes each, since numbers up to 255 must be represented. The monitor is close to 1K bytes in length, so just converting to decimal would multiply the memory required to put the monitor in BASIC DATA statements to probably 2.5K.

Then, an area must be set aside to store the monitor as it is POKEd into memory, so we've run out of memory for the logic required in the BASIC program for reading the DATA statements, doing the POKEing, etc. One way to reduce the memory required is to store the machine language as hexadecimal numbers in the BASIC program. This helps somewhat because each machine code could be stored in only two bytes. But this adds to the BASIC program logic overhead, since we must now include a hex-to-decimal conversion subroutine for use with the POKE statements. And there is still another problem not solved.

TINYMON1 will be placed in the same memory area that a BASIC program would occupy. So, we would be POKEing the code atop the BASIC program that was doing the POKEing. Another possibility would be to enter the program byte-by-byte, using direct entry statements. This would be awfully slow, since each byte would have to be converted to decimal, then entered from the keyboard using POKE statements. You would have to keep up with the current address, too. So this doesn't seem very attractive.

What about a BASIC program that allows the VIC to act as a loader for entering TINYMON1? A BASIC program could very easily accept a hex number input from the keyboard, convert it to decimal, POKE it in place, increment the address, then ask for the next entry. This seemed like a reasonable approach, but there was still a conflict in memory locations. Any BASIC program entered would occupy the same memory locations that the start of TINYMON1 occupies. Of course, if the BASIC program were small, then most of TINYMON1 could be entered using the BASIC "loader," with the rest entered using the direct entry method. This, then, was to be my way of entering TINYMON1. But before I could do anything, I had to have a way to SAVE all of my efforts on cassette tape.

SAVEing It

It turned out that SAVEing the program was an easy problem to solve. Simply looking through the memory map, I found several memory locations of interest. In order for the system to keep track of how its memory is used, many locations contain address information that points (called *pointers*) to the beginning and end of certain memory allocations. A quick look at the memory map located two useful pointers.

Locations 43 and 44 (in decimal) point to the start of the BASIC program, with the low order byte in 43, the high order byte in 44. Locations 45 and 46 point to the end of the BASIC program. But when I altered the end of BASIC pointer and tried a SAVE, the special test bytes I had entered into memory were not restored when I read the tape back in. So, back to the memory map. Sure enough, at locations 174 and 175 was a pointer for the tape ending address. I tried the test bytes again, and they were loaded back in. All I had to do, then, was properly set up these three pointers, and I could save any area of memory.

If At First You Don't Succeed

I was pretty confident that I had all of the keys needed to now enter TINYMONT directly into the VIC. Several nights later, after all entries were made and all systems appeared go, I anxiously entered RUN. My heart sank when I saw an ILLEGAL QUANTITY error flash onto the screen. No amount of rechecking or rethinking did any good. It just didn't work.

After several more nights of discouragement, I put it all away, admitting defeat. Apparently, there was something about the internal operating system I didn't know, and my attempt to enter the program violated the VIC's integrity. I had all but forgotten about the program until, during one sleepless night, I resorted to reading the corrections column in the March 1982 **COMPUTE!** *[Each month, any corrections or improvements to previously published programs are described on the CAPUTE! page. See the Table of Contents. – Ed.]* Eureka! Butterfield pointed out an error in his article on TINYMONT! Hoping against hope, I made the correction in my version, tried it, and it worked!

Now for the details. First, enter the following program into your VIC *just as listed*. Add no spaces, or it will be too long.

5 INPUTS

10 PRINTS: INPUTA\$:B(1)=0:B(2)=0

CHAPTER SIX

```
20 FOR I=1 TO 2: B(I)=ASC(MID$(A$,I,1))-48: IF B(I)  
    >9 THEN B(I)=B(I)-7  
30 NEXT  
40 POKES,B(1)*16+B(2): S=S+1: GOTO 10
```

This is the BASIC “loader.” Line 5 gets the starting decimal address of the data entry. Line 10 prints the current decimal address, then gets the data byte, which is entered in hexadecimal form. The conversion to decimal is then done in lines 20 and 30. Line 40 POKES the data into the current address, the address is incremented, and the process continues. Make a copy of this program before continuing. Any time POKES are to be performed, *never* test anything until your latest version is safely committed to tape. A misplaced bit may lock up the VIC, with no way to recover short of turning off the power, thus losing your program. After you’ve made a copy, try running it, and confirm that it works by entering data into unused RAM (location 5000 decimal is a good place) and verifying that the proper things are POKEd into the right places. Since no provision is made for exiting the program, just hold down the RUN STOP key, then push the RESTORE key when you are done. Next, make these entries directly from the keyboard:

```
POKE 45,20  
POKE 46,20  
POKE 174, 20  
POKE 175, 20
```

The VIC thinks that the end of your BASIC program has now been extended to 5140 decimal ($20 \times 256 + 20 = 5140$). Just to verify that you have succeeded in tricking the VIC, try the following. First, type in:

```
POKE 5139, 255
```

Then, save the program as you usually would. It should take somewhat longer to save than before, since you should now be recording data up to location 5139. Turn the VIC off and on again, type in:

```
POKE 5139, 0
```

and then reload your program. If typing in ? PEEK (5139) results in 255, then you’ve done fine so far. Otherwise, recheck your pointers and try again. Once you’ve begun the program entry, don’t alter the “loader” in any way, since this will cause BASIC to change the values of the pointers, resulting in unknown consequences. It wouldn’t hurt to occasionally check the pointers before making a

SAVE, since that takes less time than re-entering the entire TINYMOM1 listing in case something did go wrong.

Before you begin entering TINYMOM1, a little extra groundwork will help your bookkeeping efforts. Beginning at the first of the TINYMOM1 listing, write the VIC-equivalent addresses in decimal in the left margin. Start with 4096 at 0400, and add eight to each additional line. You should end up with 5136 at the last line, given as 0810 in the program.

Time To Enter The Program

The long haul can now begin. Run the BASIC program, and enter 4232 as the starting address. (I think you should actually be able to start at 4219, but I didn't, for reasons I won't go into. But since I know 4232 works, that's where I'm going to suggest you begin.) The next prompt requires inputting of the hex number, in this case 24, since that is the byte located at 4232 in the listing, given as 0488 originally. Just keep on with each sequential entry, occasionally verifying that your present address, as displayed on the VIC, is the same as your place in the TINYMOM1 program. Note that there are eight bytes in each line, and that you should skip over the four-digit address at the first of each line.

Enter all letters as capitals, not in lowercase as shown in the program, and always enter two-digit numbers, such as 00, not 0. Also, be careful not to confuse "8"s, "9"s, and "3"s. The printing wasn't clear on a couple of "8"s, making them look like "9"s. Every once in a while, it would be a good idea to make a copy of your progress, just in case a power blip comes along. If you quit for an extended period, be sure to write down the next starting address, and begin there when you continue. Just enter that address when you run the BASIC program. If you try a ? FRE (0) command, and the VIC locks up, just do a RUN STOP/RESTORE. The FRE (0) command sometimes doesn't work, but I've had no trouble once the TINYMOM1 entry was complete.

Make your last entry at 5139. If you go past 5139, you will be imposing on the VIC's array storage area. The last four "00"s in the listing are not important to TINYMOM1 anyway. You'll see their use later on. When you've finished these entries, then it's time to make the required direct entries. You're only 136 bytes from success!

The next thing you have to do is enter the first 136 bytes of the program using POKE statements. Since this requires decimal numbers, Figure 2 lists the first part of TINYMOM1 in decimal. Type in

POKE 4096, 0

and continue for the entire 136 entries, incrementing the POKE address by one each time. You can use the screen editing capability to avoid typing in the entire statement each time, although making each entry on a separate line might aid you in keeping up with the addresses better. Either way, it's slow, but the results will be worth it. The more astute reader might notice that the decimal listing has three entries that are different from the original; these are the changes made to the BASIC statement pointers.

Checking It Out

Once the entire listing is complete, make a copy. Now you can run the checking program, which will add up every eight bytes and display the results. By comparing the numbers displayed to those provided by Butterfield, you can locate errors, since an incorrect entry will throw off the addition. Three changes to the original addition results are necessary, though, since the BASIC pointer values were changed. The first number will be 474, the fourth 411, and the seventh 604. The rest are unchanged.

One problem with the checking program as given by Butterfield is that it quickly fills the VIC screen with numbers, making it difficult to check your program. The following direct entry program takes care of that problem, and also adjusts for the VIC memory locations we've used. Enter it directly into the VIC without using a line number. Be sure not to push RETURN until the final NEXT is entered. You'll have to use a couple of BASIC abbreviations in order to fit the program within the VIC four-line limit. Abbreviating the last two NEXTs (with N-shift E) will take care of it nicely.

Before you enter the check program, make sure the last four bytes (5140 through 5143) are 0, or the last check value displayed will be wrong. TINYMON1 doesn't actually need these bytes, and they are not saved on tape. But the checking program needs them. When using the program, just push the SPACE bar each time you want the next number. The longer you hold it down, the more numbers that will be displayed.

```
FORJ=4096TO5143STEP8:T=0:FORK=JTOJ+7:T=T+
PEEK(K):NEXT:?T:WAIT197,32,0:FORY=0TO500:N
EXT:NEXT
```

It's best to write down each check value, and after all are done, go back to the individual lines to find your mistakes. Just use the BASIC POKes and PEEKs to isolate the problems. You'll probably need to do a few hex-to-decimal conversions before you're through,

so here's a quick method. Multiply the first digit by 16, and add the second digit to the result. Remember, A = 10, B = 11, C = 12, D = 13, E = 14, and F = 15. For example, A6 hex = $(10 \times 16) + 6 = 166$ decimal.

TINYMON1 In Action

All that should be left now is to try it! Just type in RUN, hit RETURN, and the TINYMON1 should display the 6502 register contents. The other functions are described in Butterfield's article. If for some unthinkable reason it doesn't work, all I can suggest is to check all pointers, and then run the checking program again. It's possible that you made two or more errors in a line that offset each other, giving a correct check value. The only way to catch that is to inspect each byte and compare it to the original listing.

One thing I've discovered is that doing a RESTORE removes the pointer that allows re-entering TINYMON1 by doing a SYS 4096 command. This can be fixed by POKEing the correct pointer values into 790 and 791 (decimal). For a 3.5K VIC, with TINYMON1 at the top of memory, the pointer values (in decimal) are 790 = 27 and 791 = 27. The best way to make sure what these values are, in your particular application, is to use PEEK statements to see the contents of 790 and 791 after TINYMON1 has been RUN.

For VICs With More Than 8K

If your VIC has more than 8K of memory, some things change in the memory allocations. Since I don't have a VIC with all of that memory, I can only suggest the changes that you would need to make in order to use TINYMON1 on a VIC with the extra memory. The same fundamental approach should work, but the BASIC memory begins at 4608 decimal (1200 hex), so you'd have to change the BASIC program pointers. The TINYMON1 itself apparently is "relocatable," that is, it can be placed anywhere in memory with no changes. Perhaps an easier approach would be to remove any extra memory, and enter the program in a 3.5K VIC, as I've described. I suspect that the resulting tape will then load back into any VIC, regardless of its memory. With the guidelines I've given, you shouldn't have any trouble making it work.

One final thing I'd like to emphasize, especially to novice programmers, is that you should never be discouraged by the mysteries of machine language, operating systems, memory maps, etc. If you are willing to experiment a little, to study articles like this, to experiment with other people's programs, and so on, you will be able to greatly extend your programming abilities. Looking back on my experience with entering TINYMON1, I see I've learned a great

deal about the VIC, which I will be able to apply to other programming applications. Good luck in your machine language programming adventures.

Figure 1. Internal Construction of VIC BASIC Statements.

LINK TO NEXT STEP			LINE NUMBER						
7			100		PRINT	"	[CLEAR]	[DOWN]	[DOWN]
00	18	04	64	00	99	22	93	11	11
[RVS]	[RIGHT]	[RIGHT]	[RIGHT]	SPACE	T	1	N		
12	1D	1D	1D	20	54	49	9E		
Y	M	O	N	SPACE	END OF STATEMENT				
59	4D	4F	4E	20	00				

Figure 2. Decimal Equivalent of the First Part of TINYMON1.

Address				Data					
4096	0	24	16	100	0	153	34	147	
4104	17	17	18	29	29	29	32	84	
4112	73	78	89	77	79	78	32	0	
4120	49	16	110	0	153	34	17	32	
4128	74	73	77	32	66	85	84	84	
4136	69	82	70	73	69	76	68	34	
4144	0	76	16	120	0	158	40	194	
4152	40	52	51	41	170	50	53	54	
4160	172	194	40	52	52	41	170	48	
4168	55	56	41	0	0	0	234	234	
4176	165	45	133	34	165	46	133	35	
4184	165	55	133	36	165	56	133	37	
4192	160	0	165	34	208	2	198	35	
4200	198	34	177	34	208	60	165	34	
4208	208	2	198	35	198	34	177	34	
4216	240	33	133	38	165	34	208	2	
4224	198	35	198	34	177	34	24	101	

INDEX

- Accounting
 - Amortize 103-105
 - Electronic Checkbook 93-96
 - Home Calculation Six-Pack 8
 - Personal Finance 8
- Applications
 - Alphabetizing Sort 78-79
 - Double-Height Characters 24-25
 - Paddle and Keyboards 39-45
 - Timekeeping Pauses 122-124
- Apple 18
- Append 98, 106-108
- Arrays 135
 - Integer Variables For Memory Conservation 134-136 (See Also Variables)
 - Multiple Statements and Short Variable Names 132-134
- ASCII 40-41
- Assembly Language (See Machine Language) 142-143
- BASIC 7, 39, 41-43, 61, 89, 97-98, 100, 115, 125, 132, 161-169, 195-196, 203-204, 206
 - Conversion 141-143, 202-210
 - Structure 99-100, 100-101, 101-102, 104, 179-185, 186-188
- Cassette (See Tape)
 - In Overlaying 138-140
- Clock 119-124 (See Also TIS)
- Color
 - 6560 Software and Hardware 173-178
 - Keys 5, 28, 158
 - Screen/Border 28-29, 47-49, 92-93, 147, 155
 - User-Defined Functions 154-156
- Commodore Key 41
- Cursor Keys
 - In Quote Mode 113-114
- Edit Mode 164-165
- Function Keys 90-91, 101-102, 131, 158-159, 186
- Games 7, 59-66
 - Blue Meanies From Outer Space 7; Breakout 50-53; Gorf 7; Omega Race 7; Meteor Maze 61-66; Pong 48-50, 53-56; Sargon II Chess 7; Scott Adams Adventure Games 7; Sketch 28-30, 31-34; Spacewar 30, 34-38; STARFIGHT3 72-77; ZAP!! 67-71; for children, Home Babysitter 8; Count The Hearts 80-86
- GET 100-101
- Graphics 147, 148-153, 154-156
 - Custom Characters 24, 160-169
 - Double-Height 24-25
 - Fast Animation 67-71
 - Kaleidoscope 147
 - Sketch 28-30
 - High Resolution 148-153
- Hardware (See Also CPU, Disk, Printer, Tape, etc.) 174-178
- History
 - Of Computers 11-19
 - Of VIC 3-10
- INPUT 99-101, 104
- Joysticks 26-38, 59-66, 183
- Keyboards, Polled 40-43
- Languages (Also See BASIC, Machine Language)
 - BASIC 100
 - Direct Statements 20-23
- Machine Language 42-43, 45, 97, 100, 106, 142
 - Monitor, TINYMON1 195-201, 202-210
- Mainframes 17
- Memory 21-22, 173, 179-184, 186-187
 - Conservation 30-31, 133-134, 135, 138-140
 - Alternate Screens 115-118, Overlay 138-139, Reducing Programs 129-140
 - Expansion 30-31, 131-140, 151-153
 - Partitioning 97-98, 138-139
 - Maps 186-188, 189-191, 202-203, 205, 209-210

INDEX

- Modem 8-9
- Peripherals (See Disk, Printer, etc.)
 - Using Game Paddles 39-45, 46-56
 - Using Joysticks 26-27
- PET 3, 18-19
- PEEK 101-102
- Plotting 157-159
- Pointers (See BASIC Structure)
- POKE 158-159
- Printer 109-112, 130
- Renumber 125-126
- Review Mode 163-164
- ROM (Also See Memory, Memory Maps) 24
- RUN/STOP Key 42
- Screen Memory 28, 115-118, 173
- Sorting 78-79
- Sound 43, 183-184
- Special Characters
 - Double-Height 24-25
- TIS (See Clock) 119-124
- Toolkit, BASIC 98, 141
- User Friendly 4-5, 7-8
- Variables (Also See Arrays) 22-28,
 - 89-93, 99, 119, 134-137, 178
 - DIMensioned 78
 - Integer Variables 134-136
 - Short Variable Names 132-134
- VIC Chip 4, 5, 173-178, 179-185, 191

NOTES

NOTES

If you've enjoyed the articles in this book, you'll find the same style and quality in every monthly issue of **COMPUTE!** Magazine. Use this form to order your subscription to **COMPUTE!**

For Fastest Service,
Call Our **Toll-Free** US Order Line

800-334-0868
In NC call 919-275-9809

COMPUTE!

P.O. Box 5406
Greensboro, NC 27403

My Computer Is:

☐ PET ☐ Apple ☐ Atari ☐ VIC ☐ Other _____ ☐ Don't yet have one...

- ☐ \$20.00 One Year US Subscription
☐ \$36.00 Two Year US Subscription
☐ \$54.00 Three Year US Subscription

Subscription rates outside the US:

- ☐ \$25.00 Canada F=2
☐ \$38.00 Europe/Air Delivery FI=3
☐ \$48.00 Middle East, North Africa, Central America/Air Mail FI=5
☐ \$88.00 South America, South Africa, Australasia/Air Mail FI=7
☐ \$25.00 International Surface Mail (lengthy, unreliable delivery) FI=4,6,8

Name _____

Address _____

City _____

State _____

Zip _____

Country _____

Payment must be in US Funds drawn on a US Bank; International Money Order, or charge card.

☐ Payment Enclosed

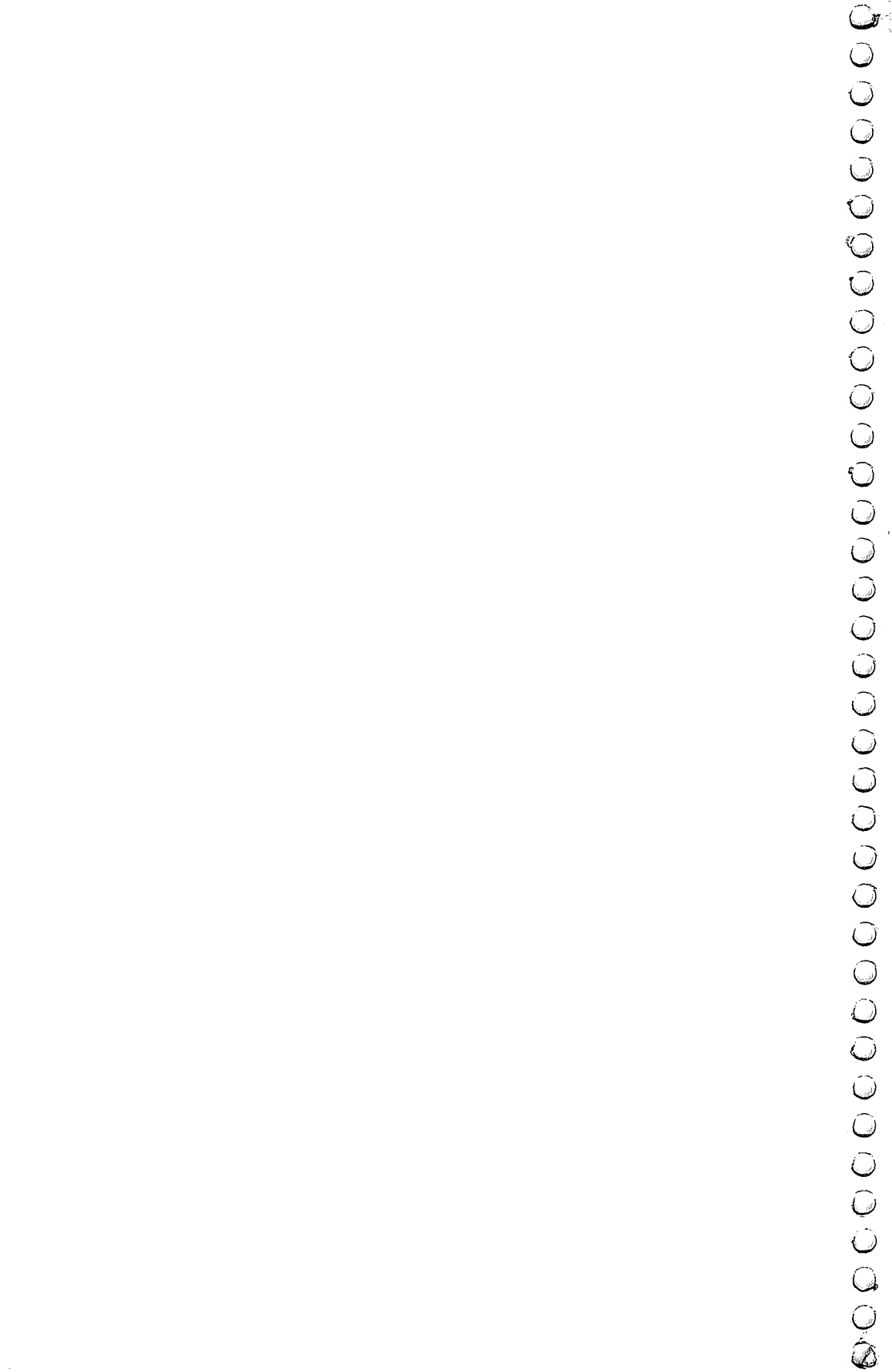
☐ VISA

☐ MasterCard

☐ American Express

Acc't. No. _____

Expires _____ / _____



COMPUTE! Books

P.O. Box 5406 Greensboro, NC 27403

Ask your retailer for these **COMPUTE! Books**. If he or she has sold out, order directly from **COMPUTE!**

For Fastest Service
Call Our **TOLL FREE US Order Line**
800-334-0868
In NC call 919-275-9809

Quantity	Title	Price	Total
_____	The Beginner's Guide To Buying A Personal Computer	\$ 3.95	_____
	(Add \$1.00 shipping and handling. Outside US add \$4.00 air mail; \$2.00 surface mail.)		
_____	COMPUTE!'s First Book of Atari	\$12.95	_____
	(Add \$2.00 shipping and handling. Outside US add \$4.00 air mail; \$2.00 surface mail.)		
_____	Inside Atari DOS	\$19.95	_____
	(Add \$2.00 shipping and handling. Outside US add \$4.00 air mail; \$2.00 surface mail.)		
_____	COMPUTE!'s First Book of PET/CBM	\$12.95	_____
	(Add \$2.00 shipping and handling. Outside US add \$4.00 air mail; \$2.00 surface mail.)		
_____	Programming the PET/CBM	\$24.95	_____
	(Add \$3.00 shipping and handling. Outside US add \$9.00 air mail; \$3.00 surface mail.)		
_____	Every Kid's First Book of Robots and Computers	\$ 4.95	_____
	(Add \$1.00 shipping and handling. Outside US add \$4.00 air mail; \$2.00 surface mail.)		
_____	COMPUTE!'s Second Book of Atari	\$12.95	_____
	(Add \$2.00 shipping and handling. Outside US add \$4.00 air mail; \$2.00 surface mail.)		
_____	COMPUTE!'s First Book of VIC	\$12.95	_____
	(Add \$2.00 shipping and handling. Outside US add \$4.00 air mail; \$2.00 surface mail.)		

All orders must be prepaid (money order, check, or charge). All payments must be in US funds. NC residents add 4% sales tax.

☐ Payment enclosed Please charge my: ☐ VISA ☐ MasterCard
☐ American Express Acc't. No. _____ Expires ____/____/____

Name _____

Address _____

City _____ State _____ Zip _____

Country _____

Allow 4-5 weeks for delivery.

